ALTIBASE Application Development

Precompiler User’s Manual

Release 5.3.3

ALTIBASE
PERFORMANCE SOLUTIONS

ALTIBASE Application Development Precompiler User’s Manual

Release 5.3.3

Altibase Corporation
10F, Daerung PostTower II, 182-13,
Guro-dong Guro-gu Seoul, 152-847, Korea
Telephone: +82-2-2082-1000 Fax: 82-2-2082-1099
E-mail: support@altibase.com www: http://www.altibase.com

Contents

Preface

About This Manual

This manual describes how to use the embedded SQL statement of Altibase and C/C++ precompiler. The user can create an application using the embedded SQL statement of Altibase and precompile the created program.

Types of Users

This manual could be useful for the following Altibase users.

- Database administrators
- Performance Managers
- Database Users
- Application designers
- Programmers
- Technical Assistamce Team

Before reading this manual, understanding of following background knowledge is recommended.

- Basic knowledge required for computers, operating systems, and operating system command
- Experience in using the relational database or understanding of the database concepts
- Computer programming experience
- Some experience with database server administartion, operating system administration or network administration

Software Environment

This manual has been prepared assuming Altibase 5.3.1 will be used as the database server.

How This Manual is Structured

This manual has been organized as follows:

- Chapter 1, "New features of preprocessor in ALTIBASE version 5.3.3."

  This chapter covers new features that preprocessor of ALTIBASE version 5.3.3. provides.

- Chapter 2, "Introduction to C/C++ Preprocessor"

  This chapter presents an introduction to C/C++ preprocessor and how to use it, and gives a detailed description of the procedure for writing program with embedded SQL statements.

- Chapter 3, "Host Variable and Indicator Variable"

This chapter sorts out host variable and indicator variable, and gives their descriptions. Additionally you can tell the value meaning of indicator variable from this chapter.

- Chapter 4, "Host Variable Declaration Section"

  This chapter explains both host variable declaration section and function argument declaration section.

- Chapter 5, "C Preprocessor"

- Chapter 6, "Data Type of Host Variable"

  This chapter covers data types used for host variable.

- Chapter 7, "Embedded SQL Statement"

  This chapter discusses statements for database connection, and DDL, DML and other embedded SQL statements.

- Chapter 8, "Handling Runtime Errors"

  This chapter explains variables referenced for handling runtime errors.

- Chapter 9, "Cursor Statements"

  This chapter describes cursor statemetns.

- Chapter 10, "Using Arrays in SQL Statements"

  This chapter covers how to use host array variable, and discusses structure, array and their restrictions.

- Chapter 11, "Dynamic SQL Statements"

  This chapter explains dynamic SQL statements.

- Chapter 12, "Usintg Stored Procedures in C/C++"

  This chapter describes how to use stored function and stored procedure.

- Chapter 13, "Multi-connection Program"

  This chapter covers how to write multi-connection program.

- Chapter 14, "Multi-thread Program"

  This chapter discusses how to write program in a multithreaded environment.

- Chapter 15, "Error Code/Message"

  This chapter explains error codes and messages.

- Appendix A, "Keyword"

  This chapter describes reserved words used in embedded SQL statements.

- Appendix B, "Restrictions on Writing Program"

This chapter covers restrictions on writing program.

- Appendix C, "Conversion between Proc* C and C/C++ Precompiler"

  This chapter discusses how to convert application program written in pro*C(C++), Oracle language, to that written in C/C++ Precompiler, ALTIBASE language.

- Appendix D, "Sample Code"

  This chapter shows example programs and explain table information related to this manual.

- Appendix E, "FAQ"

  This chapter lists frequently asked questions about how to use C/C++ Precompiler and embedded SQL statements.

Documentation Rule

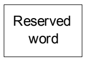
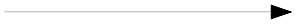
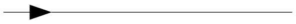
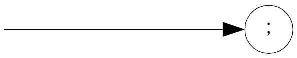
This chapter describes the rules of this manual. With understanding of this rule, it is easy to search information in this manual and other manuals.

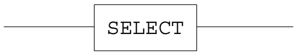
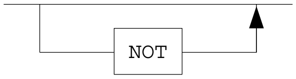
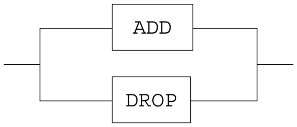
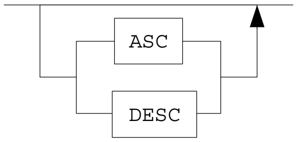
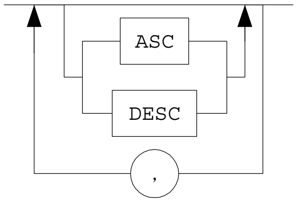
Rules are as follows:

- Syntax diagram
- Sample Program rule

Syntax Diagram

This manual describes the command syntax using the diagram composed of the following elements:

Elements	semantics
	The command starts. The syntax element which is not a complete command starts with an arrow.
	The command continues to the next line. The syntax element which is not a complete command terminates with this symbol.
	The command continues from the previous line. The syntax element which is a complete command starts with this symbol.
	End of a statement.

Elements	semantics
	Mandatory
	Optional
	Mandatory field with optional items Only one field must be provided.
	Optional field with optional item
	Optional Multiple fields are allowed. The comma must be in front of every repetition.

Sample Program Rule

The code example explains SQL, Stored Procedure, iSQL, or other command line syntax.

The following table describes the printing rules used in the code example.

Rules	Semantics	Example
[ ]	Display the optional fields.	VARCHAR [(size)] [[FIXED |] VARIABLE]
{ }	Display the mandatory fields. Display to make sure to select more than one.	{ ENABLE | DISABLE | COMPILE }

Rules	Semantics	Example
|	Argument indicating optional or mandatory fields	{ ENABLE | DISABLE | COMPILE }[ENABLE | DISABLE | COMPILE]
...	Repetition of the previous argumentDisplay the omission of the example codes.	SQL> SELECT ename FROM employee; ENAME ------------------------ SWNO HJNO HSCHOI . . . 20 rows selected.

Other symbols.	Other symbols than the above.	EXEC :p1 := 1;
		acc NUMBER(11,2);

Inclination shape	Syntax variable to be defined by the user, Location identifier for which special values must be provided	SELECT * FROM table_name;
		CONNECT userID/password;

Small letters	Program elements provided by the user such as table names, column names, file names, etc.	SELECT ename FROM employee;
Capital letters	Elements provided by the system or keyword appeared in the syntax	DESC SYSTEM_.SYS_INDICES_;

Related data

For more detailed information, see the following document list.

- ALTIBASE Administration Installation User's Manual
- ALTIBASE Administration Administrator's Manual
- ALTIBASE Application Development ODBC User's Manual
- ALTIBASE Application Development SQL User's Manual
- ALTIBASE Application Development Application Program Interface User's Manual
- ALTIBASE Tools iSQL User's Manual
- ALTIBASE Message Error Message Reference

Online Manual

Korean and English versions of on-line manuals (PDF or HTML) are available from Altibase Download

Center (http://atc.altibase.com/).

Altibase Welcomes Your Opinions!

Please send us your comments and suggestions regarding this manual. Your comments and suggestions are important, and they may be used to improve future versions of the manual. When you send your feedback, please make sure to include the following information:

- The name and version of the manual in use
- Your comments or suggestions regarding the manual
- Your name, address, and phone number

Please send your e-mail to the following address:

support@altibase.com

This address is intended to report any errors or omissions discovered in the manual. When you need an immediate assistance regarding technical issues, please contact Altibase Customer Support Center.

We always appreciate your comments and suggestions.

Part I

1 New Features of Precompiler in ALTIBASE 5.3.3.

This chapter covers new features which precompiler provides in ALTIBASE 5.3.3. Database is upgraded from version 5.3.1. to version 5.3.3. For this reason, there also has been an improvement in precompiler which database provides. You must use APRE*C/C++ precompiler in ALTIBASE version 5.3.3. or later instead of SES C/C++ precompiler in ALTIBASE version 5.3.1. or earlier.

APRE*C/C++ Precompiler

This section presents an introduction to new features of APRE*C/C++ precompiler which ALTIBASE 5.3.3. provides.

Here are useful tips for you who have intention of migrating.

New Features

C Parser

This enables you to use entire variables declared in both internal and external declaration sections as host variables.

C Preprocessor

This enables you to write conditionals such as #define, #if and #ifdef in the C preprocessor to define MACRO. For more details, see chapter 5, "C Preprocessor".

Using DECLARE STATEMENT

You can use EXEC SQL DELCARE <statement name> STATEMENT to declare identifiers in a certain sql statement or a PL/SQL block. For more details, see chapter 10, "Dynamic SQL Statements"

Using DO <function name> in WHENEVER statement

You can call a certain function every time an SQL error occurs by using DO <function name> in Whenever statement. For more details, see chapter 8, "Handling Runtime Errors".

Command Line Options

-D

This chooses macro name which should be recognized within the codes.

-keyword

This prints reserved word.

-parse

This sets the range where precompiler parses.

-I

This specifies include path.

-debug

This runs for debugging, and prints information about host variables and macro names. For more details about command line options, see command line options section in chapter 2, "Introduction to C/C++ Preprocessor".

Functional Alterations

Deleting SES_DECLARE

You can't declare host variable by writing #ifdef SES_DECLARE within #include header file, because using SES_DECLARE is not available any more in subsequent release from starting with version 5.3.3. of ALTIBASE. To achieve this, you should specify -parse as full or use EXEC SQL BEGIN/END DECLARE SECTION instead. For details, see command line options section in chapter 2, "Introduction to C/C++ Preprocessor"

Removing Restrictions on Host Variable

You can set value in host variable declaration section. If host variables are used as structures, structures can be defined after specifying data type as typedef. If host variables are used as arrays, you can set elements of arrays.

You can use char, structure or other data types as pointer host variable. Host variable without colon (:) can be used in into clause of select statement, and you can also use union variable as host variable. For more details, see chapter 4, "Host Variable Declaration Section" and chapter 6, "Data Type of Host Variable"

Changing Name of Data Type

SES_CLOB, SES_BLOB, SES_BINARY, SES_BYTES, SES_NIBBLE are changed into APRE_CLOB, APRE_BLOB, APRE_BINARY, APRE_BYTES, APRE_NIBBLE respectively. It's also safe to use existing names for compatability.

Changing Name of Executable File and Library

The executable name is altered from sesc to apre, and libapre.a also substitues for libsesc.a. However, you would have great relief of these changes, because apre and libapre.a contains duplications of sesc and libsesc respectively without modifying existing makefile.

2 C/C++ Precompiler

Introduction and Concepts

Introduction

Altibase C/C++ Precompiler is a programming tool that enables the user to embed SQL statements in a source application program.

The precompiler accepts the source program as input, translates the embedded SQL statements into standard runtime library calls, and generates a modified source program that you can compile, link, and execute in the usual way. ALTIBASE Embedded SQL C/C++ Precompiler lets you use the power and flexibility of SQL in your application programs.

Compared to creating a program using the ODBC API, the user can more easily create a program with same functions using the embedded SQL statement.

Configuration

To compile and link the output file precompiled by C/C++ Precompiler, set the configuration as follows:

Header File

The necessary header file is ulpLibInterface.h and it is stored under $ALTIBASE_HOME/include directory. To compile the precompiled application program: –I $ALTIBASE_HOME/include options are required.

Library

The necessary libraries are libapre.a and libodbccli.a and it is stored under $ALTIBASE_HOME/lib directory.

To link the pre-processed application program needed following options:

–L $ALTIBASE_HOME/lib and –lapre, –lodbccli,-lpthread

Restriction

ALTIBASE client library doesn't use signal processor.

Therefore, if access to network terminates due to external factors, application can be shut down compulsorily by receiving signal of SIGPIPE.

You may process it in user application to avoid forced shutdown.

And you can't call functions of ALTIBASE client library to process it because program can be stopped. However, you can after processing it.

Precompiling

C/C++ Precompiler precompiles a program written by C or C++ including the embedded SQL statement, and creates a converted C or C++ program. The input file has .sc extension and is made by C or C++ program. The output file has .c or .cpp extension. The user can define the extension of the output file, and the default extension is .c.

Execution Command

```
apre [ <apre-options> ] <filename>
```

Execution Argument

<filename> : A source program including the embedded SQL statement. The extension must be .sc. You can set more than 1, and then execute all of them.

[Example 1] The following example shows the command to precompile a program written in C. After precompiling, sample1.c file will be created.c.

```
shell> apre sample1.sc
```

[Example 2] The following example shows the command to precompile several programs written in C.

```
shell> apre sample1.sc sample2.sc
shell> apre *.sc
```

<apre-options> : This denotes command line options of APRE*C/C++. For details, see next section, “Command Line Options”.

Precompile Messages

The execution screen of the C/C++ precompiler is as follows:

```
shell> apre sample1.sc
--------------------------------------------------------
 APRE C/C++ Precompiler Ver 5.3.3.30
 Copyright 2009, ALTIBase Corporation or its subsidiaries.
 All rights reserved.
```

Command Line Options

전처리시 명령행에 입력할 수 있는 여러 옵션들이 있다 . 이 명령행 옵션들의 기능에 대한 설명은 아래와 같다 .

Execution Options

-h

: APRE 실행 방법을 보여준다 . 보여주는 화면은 다음과 같다 .

```
% apre -h
=============================================
 APRE (Altibase Precompiler) C/C++ Precompiler HELP Screen
==============================================
Usage : apre [<options>] <filename>
-h                : Display help information.
-t <c|cpp>        : Specify file extension for output file.
                    c - File extension is '.c' (default)
                    cpp - File extension is '.cpp'
-o <output_path> : Specify a directory path for output file.
                    (default : Current directory)
-mt               : Specify when multithreaded applications.
-I<include_path>
                  : Specify the directory paths for files included
                    using APRE C/C++. (default : Current directory)
-parse <none|partial|full>
                  : Control which non-SQL code is parsed.
-D<define_name>
                  : Define a preprocessor symbol.
-v                : Show APRE version.
-n                : Specify when unused null padding at CHAR data-type.
-unsafe_null      : Specify when allowing null fetch without indicator.
-align            : Specify when using the align from AIX.
-spill <values>   : Specify register allocation spill area.
-keyword          : Display all reserved keywords.
-debug <macro|symbol>
                  : Use for debugging.
                    macro - Display macro table.
                    symbol - Display symbol table.
-nchar_var <variable_name_list>
                  : Set client nchar variables.
-nchar_utf16      : Set client nchar encoding to UTF-16
=============================================
```

-t <c|cpp>

This defines the extension of the file created as a result of precompiling by C/C++ precompiler. When the extension is "c", .c file will be created, or when the extension is "cpp", .cpp file will be created. If no extension is specified, .c file will be created.

Example

The following example shows the command to precompile a program written in C: After precompiling, sample1.c file will be created.

```
shell> apre –t cpp sample1.sc
```

-o <output_path>

This defines the position of the file precompiled by the C/C++ precompiler. If nothing is specified, the result file will be created in the current directory. The user can define only one path under which the result files will be stored.

Example

This sets the path of a file generated by the precompilation tool with using –o. The file, sample1.c is created in ./src directory.

```
shell> apre -o ./src sample1.sc
```

-mt

In case the file to be precompiled is a multi-threaded program, this option must be selected. If the program consists of more than one file, this option must be selected for precompiling of each file. This option can be replaced with "EXEC SQL OPTION(THREADS=TRUE);" embedded SQL statement. If the above SQL statement is declared in the file to be precompiled, this option can be omitted for precompiling. For more information about OPTION statement, see Chapter V.

Example

The following example shows the command to precompile a program written in C: After precompiling, sample1.c file will be created.

```
shell> apre -mt sample1.sc
```

-I<include_path>

This defines the location of the header file to be used for precompiling. Each position must be divided by comma. If nothing is specified, the header file will be retrieved from the current directory for precompiling. Both the absolute path and the relative path can be selected.

Example

The following example shows how to indicate location of header files to be used for precompiling: The header file for precompiling will be searched in the current directory and in /include directory in order. If there is one or more directories as shown in the example, each directory must be separated by comma. Both the absolute path and the relative path can be used.

```
shell> apre –I. -I/include sample1.sc
```

-parse <none|partial|full>

전처리시 소스파일에 대한 파싱 처리 범위를 결정한다 . 위 옵션에 따라 #include 헤더파일의 처리

범위도 달라진다 . -parse 옵션을 주지 않았을경우 partial 로 처리된다 .

none

EXEC SQL BEGIN/END DECLARE SECTION 안에 있는 호스트 변수 선언들과 매크로 명령들을 처리하며 , 외부에 선언된 변수들과 매크로들은 무시된다 . 소스파일 내의 내장 SQL 구문들은 모두 처리된다 .

partial

모든 매크로 명령들을 처리하며 , 호스트 변수는 EXEC SQL BEGIN/END DECLARE SECTION 안에 선언된 것들만 처리된다 . #include 로 포함된 헤더파일은 매크로 명령들만 처리된다 . 반면 소스파일 내의 내장 SQL 구문들은 모두 처리된다 .

full

전처리기에 내장된 C 파서가 동작하여 EXEC SQL BEGIN/END DECLARE SECTION 밖에 선언된 변수들도 처리가 되며 , 모든 매크로 명령들이 처리된다 . 또한 , #include 로 포함된 헤더파일도 매크로 명령들뿐 아니라 변수 선언부분도 처리 된다 . 마지막으로 소스파일 내의 내장 SQL 구문들이 모두 처리된다 .

주의사항

-parse 옵션을 full 로 설정했을 경우 C 파서가 동작하기 때문에 C++ 소스 코드는 전처리 중 파싱 에러를 발생시킬 수 있다 . 따라서 C++ 소스 코드에서는 –parse 옵션을 아예 사용하지 않던가 또는 –parse 옵션을 사용하더라도 partial/none 으로 설정해 줘야 한다 .

예제

```
shell> apre -parse none -t cpp sample1.sc
shell> apre -parse partial -t cpp sample1.sc
shell> apre -parse full -t cpp sample1.sc
```

-D<define_name>

전처리시 사용될 매크로 이름을 저장한다 . 이 명령행 옵션은 코드 내에서 #define 과 같은 기능을 한다 .

예제

sample1.sc을 전처리하기 전 ALTIBASE라는 매크로를 정의하고 싶다면 아래와 같이 명령행 옵션을 설정한다 .

```
shell> apre -DALTIBASE -t cpp sample1.sc
```

-v

This displays the version of the C/C++ precompiler.

Example

The following example shows how to check the version of the C/C++ precompiler:

```
shell> apre –v
Altibase Precompiler2(APRE) Ver.1 5.3.3.0 INTEL_LINUX_ubuntu_8.10-32bit-
5.3.3.2-debug-GCC4.3.2 (i686-pc-linux-gnu) Dec 17 2009 11:47:30
```

-n

This selects this option when the datatype of the host variable is CHAR and null padding is not made. Since, null character does not exist in the last line of the data, be cautious then using funtion that starts with “str” such as strcat, strchr, and strcpy, which does not specify its length. The size of the input host variable must be same or smaller than the size of the DB column.

Example

The following example shows the command to precompile a program written in C: After precompiling, sample1.c file will be created.

```
shell> apre -n sample1.sc
```

-unsafe_null

This stops that error may occur even though you fetch null without indicator variable. If the value of column, in which you execute SELECT or FETCH statement, is null and you don't set indicator variable, error occurs. You need not change the value of host variable with this option even if column has null.

Example

The following example shows the command to precompile a program written in C: After precompiling, sample1.c file will be created.

```
shell> apre -unsafe_null sample1.sc
```

-spill <values>

This specifies its value only when you precompile data in AIX. How to do it appears below.

#pragma options spill=<values>

Example

The following example shows the command to precompile a program written in C: After precompiling, sample1.c file will be created.

```
shell> apre -spill AIX sample1.sc
```

-keyword

This lists reseverd keywords which can not be used in C source texts and embedded SQL statements.

Example

```
shell> apre -keyword
:: Keywords for C code ::ALTIBASE_APRE APRE_BINARY APRE_BIT APRE_BLOB
APRE_BLOB_LOCATOR APRE_BYTES APRE_CLOB APRE_CLOB_LOCATOR APRE_DUPKEY_ERR
APRE_INTEGER APRE_NIBBLE APRE_NUMERIC MAX_CHAR_PTR SESC_DECLARE SESC_INCLUDE
SES_BINARY SES_BIT SES_BLOB SES_BLOB_LOCATOR SES_BYTES SES_CLOB
SES_CLOB_LOCATOR SES_DUPKEY_ERR SES_INTEGER SES_NIBBLE SES_NUMERIC SQL-
FailOverCallback SQLLEN SQL_DATE_STRUCT SQL_TIMESTAMP_STRUCT SQL_TIME_STRUCT
VARCHAR
:: Keywords for Embedded SQL statement ::ABSOLUTE ADD AFTER AGER ALL ALLOCATE
ALTER AND ANY ARCHIVE ARCHIVELOG AS ASC ASENSITIVE AT AUTOCOMMIT BACKUP BATCH
BEFORE BEGIN BETWEEN BLOB_FILE BREAK BY CASCADE CASE CAST CLEAR_RECPTRS
CLOB_FILE CLOSE COALESCE COLUMN COMMIT COMPILE CONNECT CONSTANT CONSTRAINT
CONSTRAINTS CONTINUE CREATE CUBE CURSOR CYCLE DATABASE DEALLOCATE DECLARE
DEFAULT DELETE DEQUEUE DESC DESCRIPTOR DIRECTORY DISABLE DISABLE_RECPTR DIS-
CONNECT DISTINCT DO DROP EACH ELSE ELSEIF ELSIF ENABLE ENABLEALL_RECPTRS
ENABLE_RECPTR END ENQUEUE ESCAPE EXCEPTION EXEC EXECUTE EXISTS EXIT EXTENT-
SIZE FALSE FETCH FIFO FIRST FIXED FLUSH FOR FOREIGN FOUND FREE FROM FULL
FUNCTION GOTO GRANT GROUP GROUPING HAVING HOLD IDENTIFIED IF IMMEDIATE IN
INDEX INDICATOR INNER INSENSITIVE INSERT INTERSECT INTO IS ISOLATION JOIN KEY
LAST LEFT LESS LEVEL LIFO LIKE LIMIT LOB LOCAL LOCK LOGANCHOR LOOP MAXROWS
MERGE MINUS MODE MOVE MOVEMENT NEW NEXT NOARCHIVELOG NOCYCLE NOPARALLEL NOT
NULL OF OFF OFFLINE OLD ON ONERR ONLINE ONLY OPEN OPTION OR ORDER OTHERS OUT
OUTER PARALLEL PARTITION PARTITIONS PREPARE PRIMARY PRIOR PRIVILEGES PROCE-
DURE PUBLIC QUEUE RAISE READ REBUILD RECOVER REFERENCES REFERENCING RELATIVE
RELEASE RENAME REPLACE REPLICATION RESTRICT RETURN REVERSE REVOKE RIGHT ROLL-
BACK ROLLUP ROW ROWCOUNT ROWTYPE SAVEPOINT SCROLL SELECT SENSITIVE SEQUENCE
SESSION SET SETS SOME SPLIT SQLCODE SQLERRM SQLERROR SQLLEN START STATEMENT
STEP STORE SYNONYM TABLE TABLESPACE TEMPORARY THAN THEN THREADS TO TRIGGER
TRUE TRUNCATE TYPE TYPESET UNION UNIQUE UNTIL UPDATE USER USING VALUES VAR-
CHAR VARIABLE VIEW VOLATILE WAIT WAKEUP_RECPTR WHEN WHENEVER WHERE WHILE WITH
WORK WRITE
```

-debug <macro|symbol>

디버깅을 하기 위한 용도로 쓰이며 , 매크로 이름이나 선언된 변수의 정보를 갖고 있는 심볼테이블 전체를 출력한다 .

macro

정의된 매크로 이름의 정보를 저장하고 있는 매크로 목록을 출력한다 .

symbol

선언된 변수들의 정보를 저장하고 있는 예약어 목록을 출력한다 .

예제

sample1.sc 안에 정의된 모든 매크로를 보여준다 .

```
shell> apre -debug macro sample1.sc
```

sample1.sc 안에 선언된 변수의 정보를 보여준다 .

```
shell> apre -debug symbol sample1.sc
```

매크로와 변수 모두 출력한다 .

```
shell> apre -debug macro symbol sample1.sc
```

-nchar_utf16

This encodes character with national character type in UTF-16 when you precompile data. Data are encoded in some format specified in ALTIBASE_NLS_USE by default without this option.

However, this way can’t compute the entire unicode character and occurs data loss.

Example

The following example shows how to precompile program with UTF-16.

```
shell> apre -nchar_utf16 -t cpp sample.sc
```

-nchar_var <variable_name_list>

This option enables precompiler to process national character typed data which ALTIBASE supports. You can’t set blank between variable names, and variable name in structure.

Examples

You should specify the following option if data type of var1 and var2 used in sample1.sc is national character type.

```
shell> apre -nchar_var var1,var2 sample1.sc
```

Embedded SQL Programming

This chapter briefly describes how to write a program using embedded SQL statement and the general flow.

The following describes the method and the order of writing a program and shows the general flow of program:

Declaration of the Host Variables

When writing a program, user must declare the host variables.

The host variable must be declared in DECLARE section of the host variable.

For more information about the host variable and DECLARE section of the host variable, see Chapters II ~ IV.

Example

The following is an example of declaring the host variable:

```
< Example Program : insert.sc >
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
char usr[10];
char pwd[10];
char s_gno[10+1];
char s_gname[20+1];
char s_goods_location[9+1];
int s_stock;
double s_price;
EXEC SQL END DECLARE SECTION;
```

Connecting to a Database Server

After declaring the host variable, connect to the database server before executing another embedded SQL statement.

All embedded SQL statements must be executed after successful connection with database server.

For more information about the connection with the database server, see Chapter VII.

Example

The following is an example of connecting to the database server:

```
< Example Program : connect1.sc >
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
char usr[10];
char pwd[10];
EXEC SQL END DECLARE SECTION;
/* set username */
strcpy(usr, "SYS");
```

```
/* set password */
strcpy(pwd, "MANAGER");
EXEC SQL CONNECT :usr IDENTIFIED BY :pwd;
if (sqlca.sqlcode == SQL_SUCCESS) /* check sqlca.sqlcode */
{
   printf("Success connection to ALTIBASE server\n\n");
}
else
{
   printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
   exit(1);
}
```

Executing Embedded SQL Statements

Execute embedded SQL statement after successful connection with the database server. Embedded SQL statements includes DML such as SELECT and INSERT, DDL such as CREATE , system control statements, cursor-related SQL statements, dynamic SQL statements, and other embedded SQL statement that can execute Altibase SQL statement.

For more information about using various embedded SQL statement, see Chapters V, VII, IX, and X.

Example

Display examples of using various embedded SQL statements.

[Example 1] The following is an example of UPDATE statement:

< Example Program : update.sc >

```
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
short s_dno;
varchar s_emp_job[15+1];
EXEC SQL END DECLARE SECTION;
s_eno = 2;
s_dno = 1001;
strcpy(s_emp_job.arr, "ENGINEER");
s_emp_job.len = strlen(s_emp_job.arr);
EXEC SQL UPDATE EMPLOYEE
SET DNO = :s_dno,
 EMP_JOB = :s_emp_job
 WHERE ENO = :s_eno;
```

[Example 2] The following is an example of CURSOR statement.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
} department;
typedef struct dept_ind
{
 int dno;
 int dname;
 int dep_location;
 int mgr_no;
} dept_ind;
EXEC SQL END DECLARE SECTION;
```

< Example Program : cursor1.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
/* structure host variables */
department s_department;
/* structure indicator variables */
dept_ind s_dept_ind;
EXEC SQL END DECLARE SECTION;
/* declare cursor */
EXEC SQL DECLARE DEPT_CUR CURSOR FOR
 SELECT *
 FROM DEPARTMENT;

/* open cursor */
EXEC SQL OPEN DEPT_CUR;

/* fetch cursor in loop */
while(1)
{
 /* use indicator variables to check null value */
 EXEC SQL FETCH DEPT_CUR INTO :s_department :s_dept_ind;
 if (sqlca.sqlcode == SQL_SUCCESS) /* check sqlca.sqlcode */
 {
 printf("%d %s %s %d\n",
 s_department.dno, s_department.dname,
 s_department.dep_location,
s_department.mgr_no);
}
else if (sqlca.sqlcode == SQL_NO_DATA)
{
 break;
}
else
{
 printf("Error : [%d] %s\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 break;
}
}
/* close cursor */
EXEC SQL CLOSE DEPT_CUR;
```

Runtime Error Handling

Check the execution result of every embedded SQL statement. The execution result of the embedded SQL statement is stored in sqlca.sqlcode, and depending on the result it can be referred to variable such as SQLSTATE, SQLCODE, etc.

For more information about the variables that can be referred to check the execution result of the embedded SQL statement, see chapter VI.

Example

The following is an example of checking the execution result of the embedded SQL statement:

```
< Example Program : delete.sc >
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
short s_dno;
EXEC SQL END DECLARE SECTION;
s_eno = 5;
s_dno = 1000;
EXEC SQL DELETE FROM EMPLOYEE
WHERE ENO > :s_eno AND
DNO > :s_dno AND
EMP_JOB LIKE 'P%';
/* check sqlca.sqlcode */
if (sqlca.sqlcode == SQL_SUCCESS)
{
 /* sqlca.sqlerrd[2] holds the
rows-processed(deleted) count */
 printf("%d rows deleted\n\n", sqlca.sqlerrd[2]);
}
else
{
 printf("Error : [%d] %s\n\n",
SQLCODE, sqlca.sqlerrm.sqlerrmc);
}
```

Disconnecting from a Database Server

Disconnect from the database server after executing all embedded SQL statements before terminating the program. When the database server is disconnected, all resources allocated to the corresponding connection will be released. After the database server is disconnected, the embedded SQL statement cannot be executed.

For more information about disconnection from the database server, see Chapter VII.

Example

The following is an example of disconnecting from the database server:

```
< Example Program : connect1.sc >
EXEC SQL DISCONNECT;
```

How to Precompile

Precompiling procedure using the precompiler:

```
apre [<apre - option>] <filename>
```

Example

The following is an example of precompiling connect1.sc file:

```
Shell> apre connect1.sc
```

3 Host Variable and Indicator Variable

Host Variables

Definition

The host variable is responsible for data exchange between the program written in the host language and the database server. In other words, it stores the column data of the table in the host variable or inserts the host variable value into the column.

Declaration

The host variable can be declared as follows:

- Declare in the host variable declaration section or the function factor declaration part.

  If the embedded SQL statement uses a variable that is not declared in DECLARE section of the host variable or function argument, "Not defined host variable" error will occur upon precompiling.

  For more information about the host variable declaration part or the function factor declaration part, see Chapter III.

- Host variable declaration syntax is as follows:

  ```
  datatype variable_name;
  ```

  It is same as declaring the variables in C or C++ program.

  For more information about datatype of the host variable, see Chapter VI.

- In case of the array host variable, the user can make two-dimensional array declaration for the char type and the varchar type or one-dimensional array declaration for other types. For more information about array-processing SQL statement, see Chapter VIII.

- Precompiler can process national character typed data which ALTIBASE supports with reserved word as follows, and char and varchar typed data for host variable.

  ```
  character set [is] nchar_cs
  ```

  However, precompiler can without reserved word if you set nchar_var in command line.

Usage

The host variable can be used in a position where the scholar expression is allowed in the embedded SQL statement.

The host variable must be identified from other SQL elements in a embedded SQL statement and must have a prefix of ":".

Example

In the following example, s_dno, s_dname, and s_dep_location are declared as host variables:

```
< Example Program : select.sc >
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
char s_dname[30+1];
char s_dep_location[9+1];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT DNAME, DEP_LOCATION
INTO :s_dname, :s_dep_location
FROM DEPARTMENT
WHERE DNO = :s_dno;
```

Usage of Host Variables

The host variables are divided into input host variables and output host variables depending on whether it used as an input variable or output variable.

Output Variables

The output host variable is used in INTO clause of SELECT statement and FETCH statement. The result of the inquiry is stored in the host variable. It is same as the variable used in SQLBindCol () function of the ODBC.

Example

The following is an example of output host variable:

In this example, s_dname and s_dep_location are used as host variables. DNAME and DEP_LOCATION column data of the records meeting the conditions are stored in host variables s_dname and s_dep_location each.

```
< Example Program : select.sc >
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
char s_dname[30+1];
char s_dep_location[9+1];
EXEC SQL END DECLARE SECTION;
s_dno = 1001;
EXEC SQL SELECT DNAME, DEP_LOCATION
INTO :s_dname, :s_dep_location
FROM DEPARTMENT
WHERE DNO = :s_dno;
```

Input Variables

Input host variables are to specify input data in SQL statement. For example, it is used to specify the condition in WHERE clause of SELECT statement or the column value of the record in VALUES clause of INSERT statement.

An input host variable can be used where a scalar expression is allowed in a embedded SQL statement. However, to use a host variable under TARGET, GROUP BY or ORDER BY clause in the SELECT statement, the user should specify its type with CAST operators.

A host variable can be used with the Where clause. However, if a host variable is used for a join predicate in the Where clause, a data type cannot be known and therefore the NL join method is the only method that can be used. To eliminate this restriction, the user can use the CAST operator to determine the type of a host variable in order to choose a better join method.

Example

The following example shows various usage of the input host variable :

[Example 1] The following is an example of an input host variable in Insert statement: In this example, s_gno, s_gname, s_goods_location, s_stock, and s_price are used as input host variables. The input host variable is inserted as a column value.

```
< Example Program : insert.sc >
EXEC SQL BEGIN DECLARE SECTION;
char s_gno[10+1];
char s_gname[20+1];
char s_goods_location[9+1];
int s_stock;
double s_price;
EXEC SQL END DECLARE SECTION;
strcpy(s_gno, "F111100002");
strcpy(s_gname, "XX-101");
strcpy(s_goods_location, "FD0003");
s_stock = 5000;
s_price = 9980.21;
EXEC SQL INSERT INTO GOODS
VALUES (:s_gno, :s_gname, :s_goods_location,
:s_stock, :s_price);
```

[Example 2] The following is an example of the input host variable in UPDATE statement: In this example, s_dno, s_emp_job, and s_eno are used as input host variables. Converts DNO and EMP_JOB column values of the record meeting the conditions into s_dno and s_emp_job each.

```
< Example Program : update.sc >
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
short s_dno;
varchar s_emp_job[15+1];
EXEC SQL END DECLARE SECTION;
s_eno = 2;
s_dno = 1001;
strcpy(s_emp_job.arr, "ENGINEER");
s_emp_job.len = strlen(s_emp_job.arr);
EXEC SQL UPDATE EMPLOYEE
SET DNO = :s_dno,
 EMP_JOB = :s_emp_job
 WHERE ENO = :s_eno;
```

[Example 3] The following is an example of an input host variable in DELETE statement: In this example, s_eno and s_dno are used as input host variables. It deletes the corresponding records using the host variable value.

```
< Example Program : delete.sc >
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
short s_dno;
EXEC SQL END DECLARE SECTION;
s_eno = 5;
s_dno = 1000;
EXEC SQL DELETE FROM EMPLOYEE
WHERE ENO > :s_eno AND
DNO > :s_dno AND
EMP_JOB LIKE 'P%';
```

[Example 4] The following is an example of the input host variable in SELECT statement: In this example, s_dno is used as an input host variable. Searches the records meeting conditions using the input host variable value.

```
< Example Program : select.sc >
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
char s_dname[30+1];
char s_dep_location[9+1];
EXEC SQL END DECLARE SECTION;
s_dno = 1001;
EXEC SQL SELECT DNAME, DEP_LOCATION
INTO :s_dname, :s_dep_location
FROM DEPARTMENT
WHERE DNO = :s_dno;
```

[Example 5] In the following example, an input host variable is used under the target clause in the SELECT statement. Here, s_call is used as an input host variable.

```
< Example Program : host_target.sc >
EXEC SQL BEGIN DECLARE SECTION;
double s_call;
EXEC SQL END DECLARE SECTION;
s_call = 0.045;
EXEC SQL SELECT principal sum * ( 1 - CAST( :s_call AS DOUBLE ) ) FROM count;
```

[Example 6] In the following example, an input host variable is used under the group by clause in the SELECT statement. Here, s_period is used as an input host variable.

```
< Example Program : host_group.sc >
EXEC SQL BEGIN DECLARE SECTION;
int s_period;
EXEC SQL END DECLARE SECTION;
s_period = 1; /* 1(month), 3(quarter year), 6(half year) */
EXEC SQL SELECT SUM(sale) FROM sales
 GROUP BY FLOOR( month / CAST( :s_period AS INTEGER ) );
```

[Example 7] In the following example, an input host variable is used for join predicate in the where clause. Here, s_diff is used as an input host variable.

```
< Example Program : host_join.sc >
EXEC SQL BEGIN DECLARE SECTION;
int s_diff;
EXEC SQL END DECLARE SECTION;
s_diff = 1;
EXEC SQL SELECT * FROM t1, t2
 WHERE t1.i1 = t2.i1 + CAST( :s_diff AS INTEGER );
```

Indicator Variable

Definition

In case the column value of the table is NULL, the host language cannot express this. Therefore, separate processing is necessary.

For processing of the NULL value, the C/C++ precompiler supports indicator variable.

An indicator variable is a variable that is used with a host variable in the embedded SQL statement to process the NULL value.

Why use indicator variables ?

- Provides a value for the programmer to judge whether the column value corresponding to the indicator variables NULL or not.

  If the input indicator variable is set as "-1" (SQL_NULL_DATA), the corresponding host variable will be processes as NULL.

  If the output indicator variable is "-1" (SQL_NULL_DATA), it means the corresponding column has returned as NULL data.

  For example, the indicator variable is used to judge whether a certain host variable in INSERT statement is NULL or not or whether the selected column is NULL or not.

- The length of the input value will be specified, or the length of the returned column will be stored.

  This applies only when the datatype of the host variable is character type or binary type.

  Specifies the length of the input value in the input indicator variable.

  In the output indicator variable, the length of the returned column value is stored.

  In case the host variable is the character type and the input or the returned column is not NULL, the indicator variable does not need to be defined.

  In case the host variable is the binary type, the indicator variable must be specified (even when the input or the returned column value is not NULL.) This is because the binary type may not end the null character, and the database needs to know the length of the input value while the user needs to know the length of the returned column.) Therefore, the indicator variable must be specified in this case. For more information about Binary type, see Chapter IV.

Declaration

Declaration method of the indicator variable is as follows:

- Declare in the host variable declaration part or the function factor declaration part.

  In case a variable not declared in DECLARE section of the host variable or function argument is used as an indicator value, "Not defined host variable" error will occur upon precompiling.

For more information about the host variable declaration part or the function factor declaration part, see Chapter III.

- The variable declaration syntax is as follows:

  datatype indicator_variable_name;

  Data Type should be int, SQLLEN types, or structure type comprised of only int and SQLLEN types.

Syntax

The indicator variable using syntaxes in the embedded SQL statement are as follows:

```
<:host_variable> [INDICATOR] <:indicator_variable>
```

The keyword "INDICATOR" can be omitted.

If the host variable is not a structure, the indicator variable must not be a structure either. However, if the host variable is a structure, the indicator variable must be a structure too.

When indicator variable should be set?

The indicator variable must be specified for the following cases:

- When the input value is set as NULL data:

  Set the indicator variable as -1 (SQL_Null_Data).

- When the column corresponding to the output host variable is not a null column:

  If the selected or the fetched column value is null although no indicator variable has been specified, the execution result of the embedded SQL statement (sqlca.sqlcode) will be SQL_SUCCESS_WITH_INFO.

- When APRE_BINARY, APRE_BLOB and APRE_BYTES types are used as input/output host variables:

  As the binary-type data may not end with a null character, the database server needs to know the length of the input data. Therefore, the length of the input data must be specified in the indicator variable. In the same way, in case of the output host variable, the database server must store the length of the returned column value in the indicator variable. For more information about APRE_BINARY, APRE_BLOB and APRE_BYTES, see chapter 6, "Data Type of Host Variable".

- When APRE_NIBBLE type is used as an output host variable: When APRE_NIBBLE type is used as an input host variable, the length of the input value is specified in the first byte of the host variable. Therefore, unless the input value is NULL, the indicator variable does not need to be specified. Although it is specified, it will not be internally referred to.

  In case APRE_NIBBLE type is used as an output host variable, the database server stores the length of the returned value (byte count) in the indicator variable and the length of the returned column (nibble count) in the first byte of the host variable. For more details about APRE_NIBBLE type, see chapter 6, "Data Type of Host Variable".

Restrictions

- When the host variable is a structure, the indicator variable must be a structure. At this time, two structures must have the same number of elements.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1;
struct tag2 { int i1_ind; int i2_ind; } var1_ind1;
struct tag3 { int i1_ind; int i2_ind;
int i3_ind; } var1_ind2;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2)
VALUES (:var1 :var1_ind1);(O)
EXEC SQL INSERT INTO T1(I1, I2)
VALUES (:var1 :var1_ind2);(X)
```

- In case the host variable is an array of a structure, the indicator variable cannot be specified. This is because of an internal regulation that requires the structure to include the host variable and the indicator when the host variable is an array of the structure.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1[10];
struct tag2 { int i1_ind; int i2_ind; } var1_ind1[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2)
VALUES (:var1 :var1_ind1);(X)
```

- If the host variable is "varchar" type and an indicator variable is specified, the specified variable will be used as an indicator variable. Otherwise, len variable, a component of "varchar" type will automatically become the indicator variable. Therefore, in this case, len variable must not be used as an indicator variable.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
varchar var1;
int var1_ind;
EXEC SQL END DECLARE SECTION;
Insert ‘TEST’ in I1 column of /* T1 table.
When var1.len is used as a indicator variable */
strcpy(var1.arr, “TEST”);
var1.len = strlen(var1.arr);
EXEC SQL INSERT INTO T1(I1)
VALUES (:var1);
/* inserting NULL in I1column in T1table,
When var1.len is used as a indicator variable */
var1.len = -1;
EXEC SQL INSERT INTO T1(I1)
VALUES (:var1);
Insert ‘TEST’ in I1 column of /* T1 table.
using var1_ind as an indicator variable */
strcpy(var1.arr, “TEST”);
var1_ind = strlen(var1.arr);
EXEC SQL INSERT INTO T1(I1)
VALUES (:var1 :var1_ind);
```

Examples

In the following example, s_goods_location_ind is used as the indicator variable of s_goods_location and s_price_ind as the indicator variable of s_price. As both indicator variables are SQL_NULL_DATA, null data will be inserted in the corresponding columns.

```
< Example Program : indicator.sc >
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
char s_gno[10+1];
char s_gname[20+1];
char s_goods_location[9+1];
int s_stock;
double s_price;
/* declare indicator variables */
int s_goods_location_ind;
int s_price_ind;
EXEC SQL END DECLARE SECTION;
/* set host variables */
strcpy(s_gno, "X111100002");
strcpy(s_gname, "XX-101");
strcpy(s_goods_location, "FD0003");
s_stock = 5000;
s_price = 9980.21;
/* set indicator variables */
s_goods_location_ind = SQL_NULL_DATA;
s_price_ind = SQL_NULL_DATA;
EXEC SQL INSERT INTO GOODS
VALUES (:s_gno,
:s_gname,
:s_goods_location :s_goods_location_ind,
:s_stock,
:s_price :s_price_ind);
```

Usages of Indicator Variables

The indicator variables are divided into output indicator variables and input indicator variables depending on whether they are used with the output host variable or input host variable.

Output Variables

If the column corresponding to the output host variable is not "NOT NULL" column, an indicator variable must be used. This is because when the selected or fetched column is null and the indicator variable is not specified, the execution result of the embedded SQL statement (sqlca.sqlcode) will become SQL_SUCCESS_WITH_INFO.

If the indicator variable is -1 (SQL_NULL_DATA), it means the null data will be returned to the column. Therefore, the output host variable is not significant (or a garbage value.) If the indicator variable is not "-1" (SQL_NULL_DATA), the corresponding column value will not be null and the corresponding column value will be stored in the output host variable. For more information about the indicator variable in this case, see "Meaning of the Indicator Variable Value" part.

Example

The following is an example of an output indicator variable.

In this example, the indicator variable of s_goods is used as s_good_ind. As s_goods is a structure, s_good_ind will be declared as structure. Two structures will have the same number of components. After executing SELECT statement, checks whether the component of s_good_ind is "-1."

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct goods
{
 char gno[10+1];
 char gname[20+1];
 char goods_location[9+1];
 int stock;
 double price;
} goods;
typedef struct good_ind
{
 int gno;
 int gname;
 int goods_location;
 int stock;
 int price;
} good_ind;
EXEC SQL END DECLARE SECTION;
```

< Example Program : indicator.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
goods s_goods;
good_ind s_good_ind;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL SELECT *
INTO :s_goods :s_good_ind
FROM GOODS
WHERE GNO = :s_gno;
/* As GNO and GNAME are "Not Null" columns, the verification of the indicator
variables will be omitted. */
if (sqlca.sqlcode == SQL_SUCCESS)
{
 if (s_good_ind.goods_location == SQL_NULL_DATA)
 {
 strcpy(s_goods.goods_location, "NULL");
 }
 if (s_good_ind.stock == SQL_NULL_DATA)
 {
 s_goods.stock = -1;
 }
 if (s_good_ind.price == SQL_NULL_DATA)
 {
 s_goods.price = -1;
 }

}
```

Input Variables

To set the NULL data as the input value, use an input indicator variable. At this time, the corresponding indicator variable must be set as -1.

If the input values is not NULL, the indicator variable does not need to be specified. However, when specifying an indicator variable, take cautions. The indicator variable differs depending on the input host variable type. For more information, see "Meaning of Indicator Variable."

Example

The following is an example of an input indicator variable.

In this example, s_goods_location_ind is used as the indicator of s_goods_location, and s_price_ind as s_price. Specifies SQL_NULL_DATA (-1) in s_goods_location_ind and s_price_ind, and inserts null in GOODS_LOCATION and PRICE column each.

```
< Example Program : indicator.sc >
EXEC SQL BEGIN DECLARE SECTION;
/* declare host variables */
char s_gno[10+1];
char s_gname[20+1];
char s_goods_location[9+1];
int s_stock;
double s_price;
/* declare indicator variables */
int s_goods_location_ind;
int s_price_ind;
EXEC SQL END DECLARE SECTION;
/* set host variables */
strcpy(s_gno, "X111100002");
strcpy(s_gname, "XX-101");
strcpy(s_goods_location, "FD0003");
s_stock = 5000;
s_price = 9980.21;
/* set indicator variables */
```

```
s_goods_location_ind = SQL_NULL_DATA;
s_price_ind = SQL_NULL_DATA;
EXEC SQL INSERT INTO GOODS
VALUES (:s_gno,
 :s_gname,
 :s_goods_location :s_goods_location_ind,
 :s_stock,
:s_price :s_price_ind);
```

Meaning of Indicator Variables

Following table describes the types and values of the indicator variable as well as the meaning of the indicator variable depending on each host variable type.

If the indicator variable is "-1", it means always NULL regardless of the condition. Otherwise, the meaning of the indicator variable will differ depending on the types of the indicator variable and the host variable. Refer to the following table when using the indicator variables.

In particular, in case of the input indicator variable, the programmer must specify the value and the precompiler or the database server internally uses the specified value. Therefore, the specified value must be correct.

Type of indicator variables	input indicator variables		Output indicator variables	
Indicator variable values Host variable type	-1	Other values than -1	-1	Other values than -1
Numeric type	Means the input value is null.	Not internally referred to. Not meaningful.	Means the returned value is null.actual host variable does not mean anything. (Garbage value)	The size of the host variable (sizeof) is stored.
Character type		Define the length of the input value (strlen).		The length of the returned value (strlen) is stored.
Date type		Not internally referred to. Not meaningful.		The size of the host variable (sizeof) is stored.
APRE_BINARY		Define the length of the input value (strlen).		The length of the returned value (strlen) is stored.
APRE_BLOB		You should set the length of input value (sqllen).		The length of returned value (strlen) is stored.
APRE_CLOB		You should specify the length of input value (sqllen).		The length of returned value (strlen) is stored.
APRE_BYTES		Define the length of the input value (strlen).		The length of the returned value (strlen) is stored.
APRE_NIBBLE		Not internally referred to. Not meaningful.		The length of the returned value (strlen) is stored.

Generally, the indicator variable is for NULL processing. However, as shown in the above table, the input indicator variable is internally referred to although its value is other than "-1." Therefore, when using an input indicator variable, its value must be accurately specified although it is not NULL.

When the input indicator variable is not "-1" and the host variable type is char type or binary type, the database server will recognize the indicator variable value as the length of the input value and process accordingly.

Sample Programs

indicator.sc

See $ALTIBASE_HOME/sample/APRE/indicator.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make indicator
shell> ./indicator
<INDICATOR VARIABLES>
-----------------------------------------------------------
[Scalar Indicator Variables]
-----------------------------------------------------------
Success insert

-----------------------------------------------------------
[Structure Indicator Variables]
-----------------------------------------------------------
GNO GNAME GOODS_LOCATION STOCK PRICE
-----------------------------------------------------------
X111100002 XX-101 NULL 5000 -1.00

-----------------------------------------------------------
 [Scalar Array Indicator Variables]
-----------------------------------------------------------
3 rows updated
3 times update success

-----------------------------------------------------------
 [Arrays In Structure]
-----------------------------------------------------------
3 rows inserted
3 times inserte success

-----------------------------------------------------------
 [Indicator Variable(.len) of VARCHAR With Output Host Variables]
-----------------------------------------------------------
v_address.arr = [Pusan University]
v_address.len = 16

-----------------------------------------------------------
 [Indicator Variable(.len) of VARCHAR With Input Host Variables]
-----------------------------------------------------------
Success update

-----------------------------------------------------------
 [Indicator Variable of DATE Type With Input Host Variables]
-----------------------------------------------------------
Success update

-----------------------------------------------------------
 [Indicator Variable of DATE Type With Output Host Variables]
-----------------------------------------------------------
d_arrival_date2 = NULL
```

4 Host Variable Declaration Section

Host Variable Declaration Section

Name, type, and length of the host variable are critical information for precompiling. Therefore, the host variable must be defined in a syntax that the C/C++ precompiler can understand, and this must be declared in the DECLARE section of the host variable.

You should declare the host variable to use in the program in DECLARE section of the host variable.

Syntax

The DECLARE section of the host variable supports two syntaxes:

```
EXEC SQL BEGIN DECLARE SECTION;
/* variable_declarations */
EXEC SQL END DECLARE SECTION;
```

The DECLARE section of the host variable begins with "EXEC SQL BEGIN DECLARE SECTION;" and ends with "EXEC SQL END DECLARE SECTION;". The host variable to be used in the program must be declared between these two statements.

This syntax can be used in the file (.sc) to be precompiled and the header file (.h) used for precompiling. However, generally you must set -parse to full without uisng host variable declaration section if the header file is run by the #include command. If the header file is used in SQL INCLUDE statement, you must use host variable delaration section.

This is because when the header file (.h) refers to both the file to precompile (.sc) and C (.c) or C++ (.cpp) file using the No. 1 syntax, an error will occur during compiling.

Scope of Host Variables

the DECLARE section of the host variable can be declared globally or locally. The variable declaration scope is similar to the case in C/C++. If multiple variables with the same name are declared in different areas, the nearest variable will be override the variable in a higher scope.

This kind of override is allowed up to 50 levels.

Example

In the following example, multiple variables with the same name are declared in different scopes: The locally declared (#2) variable is prior to global variable (#1) in myfunc () function. Therefore, for the next variable (#3), the local variable must be referred to it.

```
EXEC SQL BEGIN DECLARE SECTION;
char name[20];<- #1
EXEC SQL END DECLARE SECTION;
int myfunc(void)
{
EXEC SQL BEGIN DECLARE SECTION;
char name[20];<- #2
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1 VALUES (:name);<- #3
}
```

Restrictions

There are several restrictions related to the DECLARE section of the host variable, and the developer must be aware of following when writing a program:

- The name of the host variable must begin with an alphabet letter (a ~ z, A ~ Z) and the underbar (_), and the length must not exceed 50 characters.
- In the host variable declaration part, the user cannot set the value. In other words, initialization of the value and declaration of the variable cannot be done simultaneously.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1 = 10;(X)
EXEC SQL END DECLARE SECTION;
/* Correct Example */
EXEC SQL BEGIN DECLARE SECTION;
int var1;
EXEC SQL END DECLARE SECTION;
var1 = 10;
```

- The host variable must not be a pointer type. In exceptional cases, char* is allowed.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int* var1;(X)
int var2;(O)
char* var3;(O)
EXEC SQL END DECLARE SECTION;
```

- The indicator variable must be declared in the DECLARE section of the host variable, and is subject to above limitations:

Example

The following is an example of declaration of various host variables.

```
EXEC SQL BEGIN DECLARE SECTION;<- #1
int x, y, z;<- #2
char c1[50], c2[100]; <- #3
varchar v1[50]; <- #4
struct tag1
{
int x;
char y[50];
varchar z[50];
} st1; <- #5
struct tag1 st2; <- #6
EXEC SQL END DECLARE SECTION; <- #7
```

#1 : Indicates the beginning of the DECLARE section of the host variable.

#2 : Declares Variables X, Y, and Z of int type as host variables.

#3 : Declares variables c1 and c2 of the char type as 50 and 100 bytes each.

#4 : Declares variable v1 of the varchar type as 50 bytes.

#5 : Declares variable st1 of the structure type, and defines the tag name as tag1.

#6 : Declares variable st2 of the structure type with tag1 attached.

#7 : Indicates the end of the DECLARE section of the host variable.

Data Type Definition

In the embedded SQL statement, not only the datatypes that the embedded SQL statement supports but also the datatypes that the user defined using typedef statement can be used for the host variables.

Description

The date type definition for the host variable must written in the syntax form that the precompiler can recognize, and the date type definition must be located in the DECLARE section of the host variable. The new datatype definitions can be used as host variables.

Restrictions

In case of a definition of the structure data, the data must be defined after the structure is defined. Or both the structure and the data must be defined at the same time. In case the structure is defined after the data is defined, using the host variable of this data cause an error.

Examples

Following shows various examples of the datatype definition.

[Example 1] The following is an example of the datatype definition.

```
EXEC SQL BEGIN DECLARE SECTION;
typdef unsigned int UINT;
typdef unsigned char UCHAR;
EXEC SQL END DECLARE SECTION;
```

[Example 2] The following is an example of various datatype definitions.

1. The datatype is defined after the structure

```
EXEC SQL BEGIN DECLARE SECTION;
struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
};
typedef struct department department;
EXEC SQL END DECLARE SECTION;
```

2. The structure and the datatype are defined at the same time

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
} department;
EXEC SQL END DECLARE SECTION;
```

3. The structure is defined after the datatype Incorrect Use

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct department department;
struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
};
EXEC SQL END DECLARE SECTION;
```

Macro

For the declaration of the array-type host variable, macro (#define) can be used.

Description

When declaring the array-type host variable, the user can define the number of array elements using the macro. In this case, only the constant macro can be used.

The macro not declared in the DECLARE section of the host variable can be used.

When the macro is defined in the DECLARE section of the host variable, only the constant macro can be defined. For example, a function calling macro cannot be defined in the DECLARE section of the host variable.

Macro can be defined multiple times.

In case the host variable is declared as an array variable, user can define the number of array component using the macro, operators such as +, -, /, and * as well as the formulas such as (,).

Restrictions

Macro can be used only to define the number of elements to be listed upon declaration of the array-type host variables. For example, macro definition cannot be used in the place where the host variable can be used in the internal SQL statement.

Example

The following is an example of an incorrect macro.

```
EXEC SQL BEGIN DECLARE SECTION;
#define MAX_VALUE 20
EXEC SQL END DECLARE SECTION;
EXEC SQL DELETE FROM T1 WHERE I1 = :MAX_VALUE; (X)
```

Example

The following example shows how to use the macro and define the number of array components.

```
EXEC SQL BEGIN DECLARE SECTION;
#define MAX_LENGTH 20
#define MAX_COUNT 5
char name[MAX_LENGTH];
int cnt[MAX_COUNT*MAX_LENGTH];
int capacity[MAX_COUNT];
EXEC SQL END DECLARE SECTION;
```

DECLARE section of Function Arguments

In case the argument of the function is used as a host variable, information about the function arguments must be provided for the C/C++ precompiler. The DECLARE section of function argument sends information about the function argument to the C/C++ precompiler.

Syntax

The syntax of the DECLARE section of function argument is as follows:

EXEC SQL BEGIN ARGUMENT SECTION ;

/* Declare to use the function argument as a host variable. */

EXEC SQL END ARGUMENT SECTION;

The DECLARE section of function argument begins with "EXEC SQL BEGIN ARGUMENT SECTION;" and ends with "EXEC SQL END ARGUMENT SECTION;". The function argument to be used as a host variable must be declared between these two statements.

Declare the function argument in the same way as declaring the function (with the same name and the same type.)

Description

The DECLARE section of function argument can be used only inside the function, or inside the file (.sc) during precompiling.

The user does not need to declare a global host variable using the DECLARE section of function argument or copy the function argument to the local host variable.

Therefore, convenience of development and high performance are provided. The limitations in the DECLARE section of the host variable are also applied to the DECLARE section of function argument.

Sample Program

argument.sc

See $ALTIBASE_HOME/sample/APRE/argument.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make argument
shell> ./argument
 <ARGUMENT>
```

5 C Preprocessor

Overview

APRE*C/C++ precompiler executes commands of C preprocessor mostly.

How the C Preprocessor Works

The APRE*C/C++ preprocessor recognizes most C preprocessor commands, and effectively performs the required macro substitutions, file inclusions, and conditional source text inclusions or exclusions. The APRE*C/C++ preprocessor uses the values obtained from preprocessing. It alters the source text, and generates an output file.

Example

An example should clarify the point as mentioned above. Consider the following program fragment:

```
 #include “my_header.h”
...
#if A
char name[10];
#endif
...
```

Suppose the file, my_header.h, is in the current directory. Consider the following source text:

```
#define A 1
```

The APRE*C/C++ preprocessor reads my_header.h first, and then uses the defined value, A. The APRE*C/C++ preprocessor performs the substitution of 1 for A in the declaration when using the #if command. If the conditional is true, source text includes the declaration of name. Otherwise, it excludes the declaration of name.

C Preprocessor Directives

The C preprocessor directives that APRE*C/C++ preprocessor supports are #define, #undef, #include, #if, #ifdef, #ifndef, #else, #elif and #endif.

#define, #undef

This defines a macro name used by the APRE*C/C++ preprocessor or cancels to do.

Example

```
...
#define A
#define func()
...
#undef A
#undef func
```

The APRE*C/C++ preprocessor uses the #define command as remarked above. A and func are stored in a symbol table. The APRE*C/C++ preprocessor performs the required macro substitutions whenever A and func are used. However, if the #undef command is used, APRE*C/C++ preprocessor deletes the stored names.

#include

This reads external source files which should be used by the APRE*C/C++ preprocessor to read macro with using the #define command and variables in them. For more details, see the example of overview in this chapter.How to set -parse influences on the way to execute files with using the #include command.

#if

This enables the APRE*C/C++ preprocessor to perform the required macro substitutions, and then decides whether to precompile source text followed by them.

Example

```
#define A 1 + 1
#define B A - 2
...
#if B
int var;
#endif
...
#if defined(A)
int var2;
#endif
```

The APRE*C/C++ preprocessor substitues A-2 for B and 1+1 for A to run the first #if command. This brings 1+1-2, which is an evaluation of expression to 0. Therefore, source texts between the first #if and #endif commands are removed. The APRE*C/C++ preprocessor executes defined by running the second #if command as if the #ifdef command works.

#ifdef

This decides whether to precompile source text depending on the existence of a defined name.

Example

```
#define A
#ifdef A
int var;
#endif
...
```

int var; is included in case of precompiling because A is defined as stated above.

#ifndef

This excludes source text conditionally, and runs opposite to the #ifdef command. Source texts are precompiled if no name is defined.

Example

```
#define A
#ifndef A
int var;
#endif
...
```

int var; is excluded in case of precompiling because A is defined.

#else

Source texts are precompiled, in case an #if or #ifdef or #ifndef condition is not satisfied.

Example

```
...
#define A 0
#if A
int var1;
#else
int var2;
#endif
...
```

Source texts between the #else and #endif commands are precompiled, in case an #if condition is not satisfied.

#elif

Source texts are precompiled after performing the required substitutions for the #elif command and evaluating them, in case an #if or #ifdef or #ifndef condition is not satisfied.

Example

```
...
#define A 0
#define B 1
#if A
int var1;
#elif B
int var2;
#else
int var3;
#endif
...
```

Source texts between the #elif and #endif commands are precompiled, in case an #if condition is not satisfied and an #elif condition is satisfied.

#endif

This ends an #if or #ifdef or #ifndef command.

Restrictions on Using Preprocessor

This section covers several restrictions and directives ignored by the APRE*C/C++ Preprocessor.

Ignored Directives

Several C preprocessor directives are ignored by the APRE*C/C++ preprocessor. They are not neccessary for precompiling. For example, the APRE*C/C++ preprocessor does not use the #pragma command because it is used by C compiler. The following commands are ignored when precompiled.

#

This converts a preprocessor macro parameter to a string constant.

##

This merges two preprocessor tokens in a macro definition.

#error

This produces a compile-time error message.

#pragma

This passes implementation-dependent information to the C compiler.

#line

This supplies a line number for C compiler messages.

Restrictions on #define

There are the following restrictions on using the #define command of the APRE*C/C++ precompiler. You must not perform the required substitutions for names defined in SQL statements.

Example

```
#define RESEARCH_DEPT 40
...
EXEC SQL SELECT empno, sal
 INTO :emp_number, :salary /* host arrays */
 FROM emp
 WHERE deptno = RESEARCH_DEPT; /* INVALID! */
```

An error is raised because 40 is substitued for RESEARCH_DEPT in where clause as mentioned above.

Restrictions on #if

If using a macro function in #if, a disired result can not be returned. Try not to use it.

Example

```
#define fun(X,Y) X-Y
...
#if fun(1,1)
int var;
#else
int var2;
#endif
...
```

Restictions on #include

The APRE*C/C++ Preprocessor raises an error if header file contains embedded SQL statements to run the #include command. You must not also include a VARCHAR declaration in header file. Because the APRE*C/C++ Preprocessor can run only macro commands and C variable declaration section except embedded SQL statements and a Varchar declaration.If wanting to execute header file storing embedded SQL statements or VARCHAR declaration for running the #include command, you must use EXEC SQL INCLUDE statement. If you do this, the APRE*C/C++ Preprocessor includes headerfile in output source file (Its exetension is .c or cpp.).

Example

This section explains how to input data optionally by uisng #define and #ifdef as the following examples.If ALTIBASE is defined, host variable is declared as s_goods_alti. For the rest, host variable is declared as s_goods_ora.

Example

```
< Example Program : macro.sc > /
*********************************************************
 * SAMPLE : MACRO
 * 1. Using #define, #if, #ifdef
 ********************************************************/
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
/* define ALTIBASE */
#define ALTIBASE
int main()
{
 /* declare host variables */
 char usr[10];
 char pwd[10];
 char conn_opt[1024];
 /* structure type */
#ifdef ALTIBASE
 goods s_goods_alti;
#else
 goods s_goods_ora;
#endif
 int i;
 printf("<INSERT>\n");
 /* set username */
 strcpy(usr, "SYS");
 /* set password */
 strcpy(pwd, "MANAGER");
 /* set various options */
 strcpy(conn_opt, "DSN=127.0.0.1;CONNTYPE=1"); /* PORT_NO=20300 */
 /* connect to altibase server */
 EXEC SQL CONNECT :usr IDENTIFIED BY :pwd USING :conn_opt;
 /* check sqlca.sqlcode */
 if (sqlca.sqlcode != SQL_SUCCESS)
 {
 printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 exit(1);
 }
 /* use structure host variables */
#ifdef ALTIBASE
 strcpy(s_goods_alti.gno, "F111100010");
 strcpy(s_goods_alti.gname, "ALTIBASE");
 strcpy(s_goods_alti.goods_location, "AD0010");
 s_goods_alti.stock = 9999;
 s_goods_alti.price = 99999.99;
#else
 strcpy(s_goods_ora.gno, "F111100011");
 strcpy(s_goods_ora.gname, "ORACLE");
 strcpy(s_goods_ora.goods_location, "AD0011");
 s_goods_ora.stock = 0001;
 s_goods_ora.price = 00000.01;
#endif
```

```
 /* the select insertion useing #ifdef. */
EXEC SQL INSERT INTO GOODS VALUES (
#ifdef ALTIBASE
:s_goods_alti
#else
:s_goods_ora
#endif
);
 printf("------------------------------------------------------------------
\n");
 printf("[Structure Host Variables] \n");
 printf("------------------------------------------------------------------
\n");
 /* check sqlca.sqlcode */
 if (sqlca.sqlcode == SQL_SUCCESS)
 {
 /* sqlca.sqlerrd[2] holds the rows-processed(inserted) count */
 printf("%d rows inserted\n\n", sqlca.sqlerrd[2]);
 }
 else
 {
 printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 }
 /* disconnect */
 EXEC SQL DISCONNECT;

 /* check sqlca.sqlcode */
 if(sqlca.sqlcode != SQL_SUCCESS)
 {
 printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 }
}
```

ALTIBASE_APRE Macro

ALTIBASE_APRE is a macro defined by the APRE*C/C++ preprocessor previously, and is used to decide whether to precompile certain parts of source texts. You can use usually ALTIBASE_APRE when an unnecessary and huge header file is included in case of precompiling. The following example should clarify the point to perform the required substitutions for ALTIBASE_APRE without precompiling header.h file.

Example

```
#ifndef ALTIBASE_APRE
#include <header.h>
#endif
```

header.h file can not be read because ALTIBASE_APRE is defined while the APRE*C/C++ preprocessor precompiles source texts. However, the compiler compiles entire output file including header.h after terminating to run percompiler because the compiler does not know that ALTIBASE_APRE is already defined. You can perform the required substitutions for ALTIBASE_APRE with running #ifdef and #ifndef commands preprocessor uses.

Simple Rules

This section explains simple rules when you use APRE*C/C++ Preprocessor.

Macro Definition

If you already define a macro by using command line options of C compiler, you should also do by using -D when using APRE*C/C++.For example, if C compiler runs the following commands,

```
cc -DDEBUG ...
```

APRE*C/C++ should also define a macro as follows.

```
apre -DDEBUG ...
```

Include Path

You should set entire paths of included files used in case of precompiling only by using -I. For example, when referencing to a header file stored in /home/project/include, you should specify include path for both APRE*C/C++ and C compiler.

```
apre -I/home/project/include test.sc
cc -I/home/project/include ... test.c
```

6 Datatypes of Host Variables

Overview

Usage, features, and the declaration methods of the host variable are different from general variables in the C or C++ program. Therefore, the datatype of the host variable is different from the datatype of the general variable. This chapter describes the following:

- Datatypes of the host variable
- Extended datatype provided by the embedded SQL statement
- Relation between the column type and the host variable type

Related Issues

The following describes terms used in this chapter: Knowing these terms will help to understand this chapter.

Host Variables

The host variable is a variable declared in the DECLARE section of the host variable and used by the embedded SQL statement.

General Variables

All variables declared and used by the C or C++ program. In this chapter, it is used for the comparison purpose with the host variable for the developer's better understanding.

Host Variable Datatypes

The host variable types are the datatypes of the host variables, and include the most datatypes used in C or C++ and the extended datatypes provided by the embedded SQL statement.

General Variable Datatypes

The general datatype is used in C or C++ program. In this chapter, it is used for the comparison purpose with the host variable type for the developer's better understanding.

Column Datatypes

The column type is the datatype of the column defined in the table of the database server. The column type must be compatible with the datatype of the host variable.

Datatypes for Host Variables

The following shows the datatypes of the host variables.

- Most datatypes used in C or C++
- Extended datatype provided by the embedded SQL statement

- Datatypes declared in the DECLARE section of the host variable

General Datatypes

For datatype of the host variable, most datatypes that C or C++ supports can be used. The following describes general datatypes that can be used as the datatypes of the host variables.

Numeric Types

The following shows the numeric types that can be used as datatypes of the host variables:

Integer Type

int, short int, long int, short, long, long long, unsigned int, unsigned short int, unsigned long int, unsigned short, unsigned long, unsigned long long

Real Number Type

float, double

Restrictions

The long double type is not supported. Therefore, the long double type cannot be used as a datatype of the host variable.

Character Types

The following shows the character types that can be used as datatypes of the host variables:

Character Type

char, unsigned char

Precautions

In case the corresponding column type is CHAR type when this type is used as the output host variable, the host variable should be declared as 1-higher than colum size. As the Char type (column type) is fixed, the data as long as the column length must be returned and the null character must be stored at the last digit. In case 1 is not declared, sqlca.sqlcode value will become SQL_SUCCESS_WITH_INFO after SELECT or FETCH statement is executed.

For the host variable corresponding to the column, one host variable is used rather than the input and the output host variables are separately declared. Therefore, this type must be declared as the host variable large than the column size by 1.

Pointer Types

Pointer types that can be used as the datatypes of the host variables include char* and structure type.

char*

char* can be used as the datatype of the host variable.

Char* is convenient to use when the function argument is used as the host variable. For more information about using the host variable of the function argument, see Chapter III.

MAX_CHAR_PTR

In case of char* type, the maximum size of the host variable size must not exceed 65000.

If the size must be larger than 65000, define the size using MAX_CHAR_PTR Macro before using the char* type host variable in the embedded SQL statement.

Define MAX_CHAR_PTR macro as follows:

```
#define MAX_CHAR_PTR 90000
```

By defining MAX_CHAR_PTR macro, the user can declare char* host variable as much as the defined size or allocate it in the memory.

MAX_CHAR_PTR macro can be defined outside the DECLARE section of the host variable.

STRUCT

The pointer of the structure can be used as the datatype of the host variable. When the argument of the function is used as a host variable, the pointer type of the structure is convenient to use. For more information about the host variable of the function argument, see Chapter III.

After declaring with the pointer, it must receive proper memory space. Note that the precompiler cannot check whether memory space has been allocated or not.

Examples

[Example 1] The following example shows how to use char* type v_ename as an input host variable.

```
< Example Program : argument.sc >
void ins_employee(int v_eno, char* v_ename, short v_dno)
{
EXEC SQL BEGIN ARGUMENT SECTION;
 int v_eno;
 char* v_ename;
 short v_dno;
 EXEC SQL END ARGUMENT SECTION;
 EXEC SQL INSERT INTO TODAY_EMPLOYEE
VALUES (:v_eno, :v_ename, :v_dno);
}
```

[Example 2] The following example is the definition of MAX_CHAR_PTR.

```
#define MAX_CHAR_PTR 90000
EXEC SQL BEGIN DECLARE SECTION;
char* var1;
EXEC SQL END DECLARE SECTION;
또는
```

```
EXEC SQL BEGIN DECLARE SECTION;
#define MAX_CHAR_PTR 90000
char* var1;
EXEC SQL END DECLARE SECTION;
```

[Example 3] The following example shows various definitions of the pointer types in the structure.

1. Define the pointer type after the general structure

   ```
   struct tag1
   {
    int a;
   } *A;
   A = (struct tag1*)(malloc(sizeof(struct tag1)));
   INSERT INTO T1 VALUES ( :A ); or INSERT INTO T1 VALUES (:A->a);
   ```

2. Define the pointer type after the structure

   ```
   struct tag1
   {
    int a;
   };
   struct tag1 *A;
   A = (struct tag1*)(malloc(sizeof(struct tag1)));
   SELECT I1 INTO :A FROM T1; or SELECT I1 INTO :A->a FROM T1;
   ```

3. Define the pointer after the structure type

   ```
   typedef struct tag1
   {
    int a;
   }tag1;
   tag1 *A;
   A = (tag1*)(malloc(sizeof(tag1)));
   SELECT I1 INTO :A FROM T1; 혹은 SELECT I1 INTO :A->a FROM T1;
   ```

The following example shows how to use vDataT2, pointer type of the structure, as an input host variable.

< Example Program : pointer.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct tag
{
 char n1[11];
 int n2;
}tag;

tag *dataT2;
EXEC SQL END DECLARE SECTION;
 void ins_t2(tag* vDataT2)
{
 EXEC SQL BEGIN ARGUMENT SECTION;
 tag *vDataT2;
 EXEC SQL END ARGUMENT SECTION;
 EXEC SQL INSERT INTO T2 VALUES (:vDataT2->n1, :vDataT2->n2);
}
```

Structure Types

struct

The structure can be used as a datatype of the host variable.

The structure type removes the need to list the host variables in the embedded SQL statement when searching or inserting the entire columns of the table. The structure type enables the user to use a single host variable, which makes the development process much more convenient. For example, a host variable of the structure-type host variable can be used in VALUES clause of INSERT statement or INTO clause of SELECT statement.

Even the array in a structure of a structure of which component is the array type can be used as the datatype of the host variable. For more information about Array type, see Chapter II.

Restrictions

- When the host variable is a structure, the indicator variable must be a structure. At this time, two structures must have the same number of elements.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1;
struct tag2 { int i1_ind; int i2_ind; } var1_ind1;
struct tag3 { int i1_ind; int i2_ind;
int i3_ind; } var1_ind2;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2)
VALUES (:var1 :var1_ind1);(O)
EXEC SQL INSERT INTO T1(I1, I2)
VALUES (:var1 :var1_ind2);(X)
```

- The overlapping structure cannot be used as a host variable. In other words, the element of the structure cannot be a structure.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1
{
int i1;
struct tag2
{
int i2;
int i3;
} sub_var;
} var1;
EXEC SQL END DECLARE SECTION; (X)
```

- In case of an array of the structure, no indicator variable can be defined. Therefore, when the array type of the structure is used as an output host variable, it must be guaranteed that the returned column value is not NULL. This is because the returned column value is null. If the indicator variable is not specified, sqlca.sqlcode will be SQL_SUCCESS_WITH_INFO.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 {int i1; int i2; } var1[10];
struct tag2 {int i1_ind; int i2_ind; } var1_ind[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, i2)
VALUES (:var1 :var1_ind);(X)
```

- In case the structure array is used as a host variable in INTO clause of SELECT or FETCH statement, only one output host variable can be used. In other words, it must not be used with other host variables. Therefore, if the host variable to be used in Into clause is a structure, the number of components must be the same as the number of the columns in SELECT clause.

  In case the structure array is used as a host variable in VALUES clause of INSERT statement, only one input host variable must be used. In other words, it must not be used with other host variables. Therefore, if the host variable to be used in VALUES clause is an array type of the structure, the number of the elements of this structure must be the same as the number of columns in INSERT statement.

  ```
  Example) EXEC SQL BEGIN DECLARE SECTION;
  struct tag1 { int i1; int i2; } var1[10];
  int var2[10];
  EXEC SQL END DECLARE SECTION;
  EXEC SQL INSERT INTO T1(I1, I2, i3)
  VALUES (:var1, :var2);(X)
  ```

- The third and the fourth restrictions are due to the internal rule that requires all host variables and the indicator variables to be included in the structure when the host variable is a structure array.

Examples

The following is an example of the structure type.

After declaring the structure as the goods type, and declare s_goods of the goods type as the host variable. Then, use s_goods as the input host variable of INSERT statement.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct goods
{
 char gno[10+1];
 char gname[20+1];
 char goods_location[9+1];
 int stock;
 double price;
} goods;
EXEC SQL END DECLARE SECTION;
```

< Example Program : insert.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
goods s_goods;
EXEC SQL END DECLARE SECTION;
strcpy(s_goods.gno, "F111100003");
strcpy(s_goods.gname, "XX-102");
strcpy(s_goods.goods_location, "AD0003");
s_goods.stock = 6000;
s_goods.price = 10200.96;
EXEC SQL INSERT INTO GOODS VALUES (:s_goods);
```

Extended Data Type

The embedded SQL statement provides extended datatypes besides those that C or C++ supports. This extended datatype can be used as a datatype of the host variable. The following describes the extended datatypes and how to use them.

VARCHAR

varchar

Both varchar and VARCHAR are allowed, and it is internally a structure type. For example,

If it is declared as following:

```
varchar a[10];
```

It has the structure as following:

```
struct { int len; char arr[10] ;}a;
```

Therefore, to refer to the variable of the varchar type, indicate the component of the varchar type like a.arr.

The varchar type internally includes the indicator variable. The component, len, functions as an indicator variable. Therefore, when an indicator variable is necessary, the user does not need to specify a separate indicator variable by using a varchar type.

The varchar type internally includes an indicator variable, but a separate indicator variable can be specified. In this way, the user can specify the indicator variable that corresponds to the varchar type when declaring the indicator variable in the structure type in case the varchar type is one of structure components.

Advantages

This type includes the indicator variable so that the user does not need to specify a separate indicator variable. Therefore, it is convenient when the indicator variable is necessary.

Restrictions

Unless a separate indicator variable is specified for this type, len, one of the components, will function as the indicator variable. Therefore, if a separate indicator variable has not been specified when this is used as an input host variable, len must be specified. If NULL data is set as len, it will be "-1." Otherwise, it will be the input value (arr value.)

```
Example) EXEC SQL BEGIN DECLARE SECTION;
varchar var1;
EXEC SQL END DECLARE SECTION;
strcpy(var1.arr, “ABC”);
var1.len = strlen(var1.arr);
EXEC SQL INSERT INTO T1(I1)
VALUES (:var1);(O)
```

In case the corresponding column type is CHAR type when this type is used as the output host variable, the host variable should be declared as 1-higher than colum size. As the CHAR type (column type) is fixed, the data as long as the column length must be returned and the null character must be stored at the last digit. In case 1 is not declared, sqlca.sqlcode value will become SQL_SUCCESS_WITH_INFO after SELECT or FETCH statement is executed.

In case the varchar array is used as a host variable in INTO clause of SELECT or FETCH statement, only one output host variable can be used. In other words, it must not be used together with other host variables. Therefore, if the array type of varchar is used in INTO clause, there must be one column in SELECT clause. In case the structure array is used as a host variable in VALUES clause of INSERT statement, only one input host variable must be used. In other words, it must not be used together with other host variables. Therefore if you use varchar array type in VALUES clause, column count in INSERT statement should be 1. This is because of the limitation of the structure as the varchar is internally a structure.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
varchar var1[10];
int var2[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2, i3)
VALUES (:var1, :var2);(X)
```

Example

The following is an example of the varchar type.

The example uses the host variables of the varchar type as the input host variable and the output host variable. Anit it uses s_cus_job as the input host variable, and s_address as the output host variable. Specify the length of s_cus_job.arr in s_cus_job.len. Executes SELECT statement, and checks whether s_address.len is -1 or not.

```
< Example Program : varchar.sc >
EXEC SQL BEGIN DECLARE SECTION;
char s_cname[20+1];
varchar s_cus_job[20+1];
varchar s_address[60+1];
EXEC SQL END DECLARE SECTION;
strcpy(s_cus_job.arr, "WEBMASTER");
s_cus_job.len = strlen(s_cus_job.arr);
EXEC SQL SELECT CNAME, ADDRESS
INTO :s_cname, :s_address
FROM CUSTOMER
 WHERE CNO = BIGINT'7004052321123'
AND CUS_JOB = :s_cus_job;
```

Date Type

The date type can be used only when the column type is date.

The developer can choose one of three date types.

SQL_DATE_SURUCT

You can refer to the year, month, and date when using this type. The structure of this type is as follows:

```
typedef struct tagDATE_STRUCT {
SQLSMALLINT year;
SQLSMALLINT month;
SQLSMALLINT day;
} DATE_STRUCT;
```

Example

The following is an example of SQL_DATE_STRUCT type.

In the following example, s_date is used as input or output host variable.

```
< Example Program : date.sc >
EXEC SQL BEGIN DECLARE SECTION;
SQL_DATE_STRUCT s_date;
int s_ind;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT JOIN_DATE
INTO :s_date :s_ind
FROM EMPLOYEE
WHERE ENO = 3;
s_date.year = 2003;
s_date.month = 5;
s_date.day = 9;
EXEC SQL UPDATE EMPLOYEE
SET JOIN_DATE = :s_date
WHERE ENO = 3;
```

SQL_TIME_STRUCT

Refers to the hour, minute, and second when using this type. The structure of this type is as follows:

```
typedef struct tagTIME_STRUCT {
SQLSMALLINT hour;
SQLSMALLINT minute;
SQLSMALLINT second;
} TIME_STRUCT;
```

Example

The following is an example of SQL_TIME_STRUCT type.

In the following example, s_time is used as an input or output host variable.

< Example Program : date.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
SQL_TIME_STRUCT s_time;
int s_ind;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT JOIN_DATE
INTO :s_time :s_ind
FROM EMPLOYEE
WHERE ENO = 3;
s_time.hour = 12;
s_time.minute = 12;
s_time.second = 12;
EXEC SQL UPDATE EMPLOYEE
SET JOIN_DATE = :s_time
WHERE ENO = 4;
```

SQL_TIMESTAMP_STRUCT

Refers to the year, month, day, hour, minute, and micro minutes when using this type. The structure of this type is as follows:

```
typedef struct tagTIMESTAMP_STRUCT {
SQLSMALLINT year;
SQLSMALLINT month;
SQLSMALLINT day;
SQLSMALLINT hour;
SQLSMALLINT minute;
SQLSMALLINT second;
SQLINTEGER fraction;
} TIMESTAMP_STRUCT;
```

Example

The following is an example of SQL_TIMESTAMP_STRUCT type.

In the following example, s_timestamp is used as an input or output host variable.

```
< Example Program : date.sc >
EXEC SQL BEGIN DECLARE SECTION;
SQL_TIMESTAMP_STRUCT s_timestamp;
int s_ind;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT JOIN_DATE
INTO :s_timestamp :s_ind
FROM EMPLOYEE
WHERE ENO = 3;
s_timestamp.year = 2003;
s_timestamp.month = 5;
s_timestamp.day = 9;
s_timestamp.hour = 4;
s_timestamp.minute = 0;
s_timestamp.second = 15;
s_timestamp.fraction = 100000;
EXEC SQL UPDATE EMPLOYEE
SET JOIN_DATE = :s_timestamp
WHERE ENO = 5;
```

Binary Type

When the column type is blob, BYTE, or NIBBLE, the binary type can be the host variable type.

The binary type is internally defined as follows:

```
typedef char APRE_CLOB;

typedef char APRE_BLOB;

typedef char APRE_BINARY;

typedef char APRE_BYTES;

typedef char APRE_NIBBLE;
```

The following binary types are supported:

APRE_CLOB

You can use this only if column type is CLOB, and should set indicator variable.

When host variable with this type is set to input host variable, indicator variable should be specified as -1 if you want to set host variable to NULL, and otherwise, indicator variable should be specified as value length of host variable.

When host variable with this type is set to output host variable, NULL is retuned to host variable if indicator variable is specified as -1, and indicator variable should be set to value length of host variable if indicator variable is greater than 0.

Example

The following example shows how to use APRE_CLOB.

Input host variable is set to ins_clob and input indicator variable is set to ins_clob_ind respectively. The value of ins_clob_ind is specified as value length of ins_clob.

Output host variable is set to sel_clob and output indicator variable is set to sel_clob_ind respectively. If sel_clob is specified as NULL after executing SELECT statement, sel_clob_ind is set to -1. Otherwise, sel_clob_ind is set to value length of sel_clob.

< Example Program : binary.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
APRE_CLOB ins_clob[10+1];
APRE_CLOB sel_clob[10+1];
SQLLEN ins_clob_ind;
SQLLEN sel_clob_ind;
EXEC SQL END DECLARE SECTION;
memset(ins_clob, 0x41, 10);
ins_clob_ind = 10; /* set length of ins_clob value to indicator variable */
EXEC SQL INSERT INTO T_CLOB
VALUES (:ins_clob :ins_clob_ind);
EXEC SQL SELECT *
INTO :sel_clob :sel_clob_ind
FROM T_CLOB;
```

APRE_BLOB

You can use this only if column type is BLOB, and should set indicator variable.

When host variable with this type is set to input host variable, indicator variable should be specified as -1 if you want to set host variable to NULL, and otherwise, indicator variable should be specified as value length of host variable.

When host variable with this type is set to output host variable, NULL is retuned to host variable if indicator variable is specified as -1, and indicator variable should be set to value length of host variable if indicator variable is greater than 0.

Example

The following example shows how to use APRE_BLOB.

Input host variable is set to ins_clob and input indicator variable is set to ins_clob_ind respectively.

The value of ins_clob_ind is specified as value length of ins_clob.

Output host variable is set to sel_clob and output indicator variable is set to sel_clob_ind respectively. If sel_blob is specified as NULL after executing SELECT statement, sel_blob_ind is set to -1. Otherwise, sel_clob_ind is set to value length of sel_blob.

<Example Program : binary.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
APRE_BLOB ins_blob[10+1];
APRE_BLOB sel_blob[10+1];
SQLLEN ins_blob_ind;
SQLLEN sel_blob_ind;
EXEC SQL END DECLARE SECTION;
memset(ins_blob, 0x21, 10);
ins_blob_ind = 10; /* set length of ins_blob value to indicator variable */
EXEC SQL INSERT INTO T_BLOB
VALUES (:ins_blob :ins_blob_ind);
EXEC SQL SELECT *
INTO :sel_blob :sel_blob_ind
FROM T_BLOB;
```

APRE_BINARY

Can be used only when the column type is blob.

The indicator variable must be defined.

To define the NULL as an input value when this type of the host variable is used as an input host variable, set the indicator variable as –1. Or to set other values (except NULL) as input values, define the length of the value stored as the host variable.

If the indicator variable is –1 when this kind of the host variable is used as the output host variable, NULL will be returned. If it is larger than 0, the length of the value stored in the host value will be stored in the indicator variable.

Example

The following is an example of APRE_BINARY type.

Uses ins_blob as the input host variable, and ins_blob_ind as the input indicator variable. The length of ins_blob is stored in ins_blob_ind. Uses sel_blob as the output host variable, and sel_blob_ind as the output indicator variable. If sel_blob is NULL after SELECT statement is executed, "-1" will be stored in sel_blob_ind. Otherwise, the length of sel_blob will be stored.

< Example Program : binary.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
APRE_BINARY ins_blob[10+1];
APRE_BINARY sel_blob[10+1];
int ins_blob_ind;
int sel_blob_ind;
EXEC SQL END DECLARE SECTION;
memset(ins_blob, 0x21, 10);
ins_blob_ind = 10; /* set length of ins_blob value to indicator variable */
EXEC SQL INSERT INTO T_BLOB
VALUES (:ins_blob :ins_blob_ind);
EXEC SQL SELECT *
```

```
INTO :sel_blob :sel_blob_ind
FROM T_BLOB;
```

APRE_BYTES

Can be used only when the column type is BYTE.

The indicator variable must be defined.

To define the NULL as an input value when this type of the host variable is used as an input host variable, set the indicator variable as –1. Or to set other values (except NULL) as input values, define the length of the value stored as the host variable.

If the indicator variable is –1 when this kind of the host variable is used as the output host variable, NULL will be returned. If it is larger than 0, the length of the value stored in the host value will be stored in the indicator variable.

Example

The following is an example of APRE_BYTES type.

Uses ins_bytes as the input host variable, and ins_bytes_ind as the input indicator variable. The length of ins_bytes_ind is stored in ins_bytes. Uses sel_bytes as the output host variable and sel_bytes_ind as the output indicator variable. If sel_bytes is NULL after SELECT statement is selected, -1 will be stored in sel_bytes_ind. Otherwise, the length of sel_bytes will be stored.

< Example Program : binary.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
APRE_BYTES ins_bytes[5+1];
APRE_BYTES sel_bytes[5+1];
int ins_bytes_ind;
int sel_bytes_ind;
EXEC SQL END DECLARE SECTION;
memset(ins_bytes, 0x21, 5);
ins_bytes_ind = 5; /* set length of ins_bytes value to indicator variable */
EXEC SQL INSERT INTO T_BYTES
VALUES (:ins_bytes :ins_bytes_ind);
EXEC SQL SELECT *
INTO :sel_bytes :sel_bytes_ind
FROM T_BYTES;
```

APRE_NIBBLE

Can be used only when the column type is NIBBLE.

In case of the input host variable, use the indicator value to input null. To input other values, use the first byte of the host variable. In this case, the indicator variable will be prior. In other words, if the indicator variable is found -1, NULL will be input. Otherwise, the first byte value of the host variable will be input as the length of the input data. Therefore, to input NULL data, the indicator variable must be "-1." However, to input other values, specify the length of the input data in the first byte of the host variable.

In the first byte, the length of the input data is stored. Therefore, the actual data will be stored from the second byte of the host variable. Therefore, the length of the input data will be calculated form the second byte of the host variable and it means the nibble count. One nibble is 4 bits.

In case of the output host variable, when the indicator variable is -1, NULL data will be returned. Otherwise, the length of the data (byte count) of the host variable will be stored in the indicator variable. The length of the actual data (nibble count, 4 bits = 1) will be stored in the first byte of the host variable, and actual data will be stored from the second byte. Therefore, in case of other data than null, the relation between the indicator variable and the first byte is as follows:

```
Indicator variable = First byte/2 + 1 (First byte)
```

Example

The following is an example of APRE_NIBBLE type.

Uses ins_nibble as the input host variable. As the input value is not NULL, specify the length of the input data (nibble count) in ins_nibble[0].

Uses sel_nibble as the output host variable, and sel_nibble_ind as the output indicator variable. If sel_nibble is NULL after SELECT statement is executed, -1 will be stored in sel_nibble_ind. Otherwise, the length of sel_nibble (or byte count) will be stored. The length of the actual data (nibble count) from sel_nibble[1] will be stored in sel_nibble[0].

< Example Program : binary.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
APRE_NIBBLE ins_nibble[5+2];
APRE_NIBBLE sel_nibble[5+2];
int sel_nibble_ind;
EXEC SQL END DECLARE SECTION;
memset(ins_nibble+1, 0x21, 5);
ins_nibble[0] = 10; /* set length of ins_nibble value to ins_nibble[0] */
EXEC SQL INSERT INTO T_NIBBLE
VALUES (:ins_nibble);
EXEC SQL SELECT *
INTO :sel_nibble :sel_nibble_ind
FROM T_NIBBLE;
```

Sample Program

varchar.sc

See $ALTIBASE_HOME/sample/APRE/varchar.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make varchar
shell> ./varchar
<VARCHAR TYPE>
----------------------------------------------------
[Scalar VARCHAR]
----------------------------------------------------
s_cname = [DKHAN ]
s_address.arr = [YeongdeungpoGu Seoul]
s_address.len = [20]
----------------------------------------------------
[Array of VARCHAR]
-----------------------------------------------------------CUS_JOB
```

```
----------------------------------------------------
ENGINEER
DOCTOR
DESIGNER
ENGINEER
WEBMASTER
WEBPD
PLANER
PD
DESIGNER
NULL
MANAGER
BANKER
ENGINEER
BANKER
MANAGER
PLANER
NULL
ENGINEER
NULL
WEBMASTER
----------------------------------------------------
[Structure Included VARCHAR]
----------------------------------------------------
Success insert
----------------------------------------------------
[Array of Structure Included VARCHAR]
---------------------------------------------------
3 rows inserted
3 times insert success
```

date.sc

See $ALTIBASE_HOME/sample/APRE/date.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make date
shell> ./date
<DATE TYPE>
----------------------------------------------------
[SQL_DATE_STRUCT]
----------------------------------------------------
JOIN_DATE of ENO is 3 : 2000/1/11
----------------------------------------------------
[SQL_TIME_STRUCT]
----------------------------------------------------
JOIN_DATE of ENO is 3 : 0:0:0
----------------------------------------------------
[SQL_TIMESTAMP_STRUCT]
----------------------------------------------------
JOIN_DATE of ENO is 3 : 2000/1/11 0:0:0:0
----------------------------------------------------
[SQL_DATE_STRUCT]
----------------------------------------------------
Success update with SQL_DATE_STRUCT
1 rows updated
----------------------------------------------------
[SQL_TIME_STRUCT]
----------------------------------------------------
Success update with SQL_TIME_STRUCT
```

```
1 rows updated
----------------------------------------------------
[SQL_TIMESTAMP_STRUCT]
----------------------------------------------------
Success update with SQL_TIMESTAMP_STRUCT
1 rows updated
----------------------------------------------------
[Array of Structure Included Date Type]
----------------------------------------------------
Success insert
3 rows inserted
3 times insert success
```

binary.sc

See $ALTIBASE_HOME/sample/APRE/binary.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make binary
shell> ./binary
<BINARY TYPE>
----------------------------------------------------
[APRE_CLOB]
----------------------------------------------------
Success insert with APRE_CLOB
sel_clob = AAAAAAAAAA
sel_clob_ind = 10
----------------------------------------------------
 [APRE_BLOB]
----------------------------------------------------
Success insert with APRE_BLOB
sel_blob = !!!!!!!!!!
sel_blob_ind = 10
----------------------------------------------------
[APRE_BINARY]
----------------------------------------------------
Success insert with APRE_BINARY
sel_blob = !!!!!!!!!!
sel_blob_ind = 10
----------------------------------------------------
[APREBYTES]
----------------------------------------------------
Success insert with APRE_BYTES
sel_bytes = !!!!!
sel_bytes_ind = 5
----------------------------------------------------
[APRE_NIBBLE]
----------------------------------------------------
Success insert with APRE_NIBBLE
sel_nibble = !!!!!
sel_nibble_ind = 6
sel_nibble[0] = 10
```

Column and Host Variable Types

Each column type can use various host variable types. The following describes host variable types that can be converted according to the column type and the most suitable host variable types for each column type.

Input Host Variable

The following table lists the input host variable type that can be used for each column type. The host variable type with the minimum conversion cost is a recommended type that may improve performance.

Column Type		ConvertibleHost Variable Type	Minimum conversion costWithHost variable type
Character type	CHAR	char, varchar, short, int, long, long long, double, float, SQL_DATE_STRUCT, SQL_TIME_STRUCT, SQL_TIMESTAMP_STRUCT, APRE_BINARY	char, varchar
	VARCHAR	char, varchar, short, int, long, long long, double, float, SQL_DATE_STRUCT, SQL_TIME_STRUCT, SQL_TIMESTAMP_STRUCT, APRE_BINARY	char, varchar
Integer type	SMALLINT	char, varchar, short, int, long, long long, double, float	short
	INTEGER	char, varchar, short, int, long, long long, double, float	int
	BIGINT	char, varchar, short, int, long, long long, double, float	long, long long

Column Type		ConvertibleHost Variable Type	Minimum conversion costWithHost variable type
Real number type	NUMERIC NUMBER DECIMAL	char, varchar, short, int, long, long long, double, float	char, long, long long, float, double
	FLOAT	char, varchar, short, int, long, long long, double, float	float
	REAL	char, varchar, short, int, long, long long, double, float	double
	DOUBLE	char, varchar, short, int, long, long long,double, float	double
Date Type	DATE	char, SQL_DATE_STRUCT, SQL_TIME_STRUCT, SQL_TIMESTAMP_STRUCT	char, SQL_DATE_STRUCT, SQL_TIME_STRUCT, SQL_TIMESTAMP_STRUCT
Binary Type	CLOB	APRE_CLOB	APRE_CLOB
	BLOB	APRE_BLOB	APRE_BLOB
	BINARY	APRE_BINARY	APRE_BINARY
	BYTE	APRE_BYTES	APRE_BYTES
	NIBBLE	APRE_NIBFBLE	APRE_NIBBLE

Output Host Variable

The following table shows the output host variable types that can be used for each column type. The host variable type with the minimum conversion cost is a recommended type that may improve performance.

Column Type		Convertible Host Variable Type	Host Variable Type with Minimum Conversion Cost
Character Type	CHAR	char, varchar, APRE_BINARY	char, varchar
	VARCHAR	char, varchar, APRE_BINARY	char, varchar

Column Type		Convertible Host Variable Type	Host Variable Type with Minimum Conversion Cost
Integer Type	SMALLINT	char, varchar, short, int, long, long long, double, float, APRE_BINARY	short
	INTEGER	char, varchar, short, int, long, long long, double, float, APRE_BINARY	int
	BIGINT	char, varchar, short, int, long, long long, double, float, APRE_BINARY	long, long long
Real Number Type	NUMERIC NUMBER DECIMAL	char, varchar, short, int, long, long long, double, float, APRE_BINARY	char, long, long long, float, double
	FLOAT	char, varchar, short, int, long, long long, double, float, APRE_BINARY	float
	REAL	char, varchar, short, int, long, long long, double, float, APRE_BINARY	double
	DOUBLE	char, varchar, short, int, long, long long, double, float, APRE_BINARY	double
Data Type	DATE	char, SQL_DATE_STRUCT, SQL_TIME_STRUCT, SQL_TIMESTAMP_STRUCT, APRE_BINARY	char, SQL_DATE_STRUCT, SQL_TIME_STRUCT, SQL_TIMESTAMP_STRUCT
Binary Type	CLOB	APRE_CLOB	APRE_CLOB
	BLOB	APRE_BLOB	APRE_BLOB
	BINARY	APRE_BINARY	APRE_BINARY
	BYTE	APRE_BYTES,APRE_BINARY	APRE_BYTES
	NIBBLE	APRE_NIBBLE,APRE_BINARY	APRE_NIBBLE

In case of the output host variable, APRE_BINARY type can be used as a host variable for all column types. APRE_BINARY type stores the memory data in the host variable without converting the column data according to the corresponding type(memcpy). Therefore, to use APRE_BINARY type as the host variable, the user must understand how the data is stored in the memory for each column type and must analyze it. Like in this case, when APRE_BINARY type is used as a host variable, the user does not need to convert the type, which will increase the performance. However, the developer may find it complex. Therefore, APRE_BINARY type is not used in most cases except in Blob type (column type).

Part II

7 Embedded SQL Statements

Overview

The embedded SQL statement refers to the SQL statement included in the application.

Syntax

```
EXEC SQL … ;
```

The embedded SQL statement begins with "EXEC SQL" and ends with ";".

The user can use various SQL statement such as SELECT or UPDATE DML statement or CREATE or DROP DDL statement between "EXEC SQL" and ";".

Example

The following is an example of embedded SQL statement.

```
< SELECT statement : select.sc >
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
char s_dname[30+1];
char s_dep_location[9+1];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT DNAME, DEP_LOCATION
INTO :s_dname, :s_dep_location
 FROM DEPARTMENT
WHERE DNO = :s_dno;
< INSERT문 : insert.sc >
EXEC SQL BEGIN DECLARE SECTION;
char s_gno[10+1];
char s_gname[20+1];
char s_goods_location[9+1];
int s_stock;
double s_price;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO GOODS
VALUES (:s_gno, :s_gname,
:s_goods_location,
:s_stock, :s_price);
```

More detailed information about the syntax of each embedded SQL statement will be described later.

Static Versus Dynamic SQL Statements

The embedded SQL statements are mainly divided into two types. Depending on when the SQL statement is decided - upon programming or upon execution - the SQL statement is divided into the SQL statement and the dynamic SQL statement. This chapter describes only the static SQL statement. For more information about a dynamic SQL statement, see Chapter IX.

The embedded SQL statements can be divided as follows depending on the data-processing method and the roles:

Host Variable Declaration Section

Declares the host variable to be used for the embedded SQL statement. For more detailed information, see the Chapter III.

DECLARE section of function arguments

Declares the function argument to use as the host variable. For more detailed information, see the Chapter III.

CONNECTION-related SQL statement

Refers to the SQL statement related to connection and disconnection to/from the database.

Basic Embedded SQL Statement

Includes DML statement such as SELECT, UPDATE, INSERT, and DELETE as well as DDL statement such as CREATE, DROP, and ALTER.

Cursor Control SQL Statement

Refers to the SQL statement processing the data using the cursor. Includes cursor-defining, cursor-opening, and data-opening using the cursor, and cursor-closing SQL statements. For more detailed information, see the Chapter III.

SQL/PSM-processing SQL statement

It means stored procedure and stored function in SQL statement. Includes stored procedure/function-creating, recompiling, execution, and deletion SQL statements. For more detailed information, see the Chapter III.

Other embedded SQL statements

Refers to all Altibase SQL statements than the above. Includes the work control statement, system control statement, and transaction statements.

OPTION Statement

Refers to the embedded SQL statement related to setting various options that the C/C++precompiler provides.

Statements for Database Connection

Connection-related SQL statements includes those related to the connection to the database server. Includes CONNECT statement and DISCONNECT statement.

CONNECT

Connect to the database server.

Syntax

```
EXEC SQL CONNECT <:user> IDENTIFIED BY <:passwd>
[ USING <:conn_opt1> [ , <:conn_opt2> ] ];
EXEC SQL CONNECT <:user> IDENTIFIED BY <:passwd>
ENABLE XA RMID <:rmid>;
EXEC SQL CONNECT <:user> IDENTIFIED BY <:passwd>
ENABLE XA;
```

Argument

<:user> : User name to connect to the database server.

<:passwd> : User password to connect to the database server.

<:conn_opt1> : Specifies the connection method to the database server.

- DSN : IP address of the database server to connect.

- CONNTYPE : Communication method with the database server.

1 : TCP/IP

2 : UNIX DOMAIN

3 : IPC

- PORT_NO : Port number to connect to the database server.

- NLS_USE : Sets language.

KO16KSC5601 : Korean

US7ASCII : English

MS949

BIG5

GB231280

UTF8

SHIFT-JIS

EUC-JP

BATCH : Specifies the Batch Processing Mode of the session to connect.

ON : Batch Processing Mode

OFF : Non Batch Processing Mode

<:conn_opt2> : The connection method-specifying method is same as conn_opt1. Automatically connect to the database server using conn_opt2 when the connection with the database server using conn_opt1 fails.

<:rmid> : Establishes XA connection in distributed transaction environment, but doesn't create a new connection actually. XA connection specified in rmid host variable is used for subsequent embedded SQL statements.

Description

Allow one or more connections within one program of the embedded SQL statement. In this case, only one connection without the connection name is allowed. In this chapter, only this connection (without the connection name) will be covered.

For more information about multi-connection program or multi-threaded program, see Chapters XI and XII.

* Notes: If PORT_NO and NLS_USE are not indicated in the connection string, the same value set in the property file must be set using the next environment variable.

```
export ALTIBASE_PORT_NO=20300
export ALTIBASE_NLS_USE=US7ASCII
```

The execution result when two connection options are specified

SQL_SUCCESS : When connection is established with the first option

SQL_SUCCESS_WITH_INFO : When connection is established with the second option after failing with the first option The first connection failure error message is stored in sqlca.sqlerrm.sqlerrmc.

SQL_ERROR : When connection is not established with both options The error messages are consecutively stored in sqlca.sqlerrm.sqlerrmc.

Precautions

If connection is attempted after a connection is established, "Already connected" error message will be displayed. Therefore, to establish a connection while a connection exists, execute Free or Disconnect first. At this time, if the database server is running, execute DISCONNECT. Otherwise, perform FREE.

When setting CONNTYPE as 2 or e while defining the connection method with USING clause, DSN and PORT_NO options will be ignored although they are set. Instead, the connection with the database server will be attempted.

Example

Shows various examples of connecting to the database server.

[Example 1] The following example is to connect to the database server using the user name and the user password. Connects to the database server by referring to altibase.properties file for the user name, the user password, and other necessary information.

```
< Example Program : connect1.sc >
EXEC SQL BEGIN DECLARE SECTION;
char usr[10];
char pwd[10];
EXEC SQL END DECLARE SECTION;
strcpy(usr, "SYS");
strcpy(pwd, "MANAGER");
EXEC SQL CONNECT :usr IDENTIFIED BY :pwd;
```

[Example 2] The following example is to set the connection method in USING clause and to connect to the database server. Connects to the database server using the user name and the user password stored in user and pwd and connection information stored in conn_opt3. At this time, connection information not stored in conn_opt3 will be retrieved from altibase.properties file.

```
< Example Program : connect1.sc >
EXEC SQL BEGIN DECLARE SECTION;
char usr[10];
char pwd[10];
char conn_opt3[100];
EXEC SQL END DECLARE SECTION;
strcpy(usr, "SYS");
strcpy(pwd, "MANAGER");
strcpy(conn_opt3, "DSN=192.168.11.12;CONNTYPE=1;PORT_NO=53000");
EXEC SQL CONNECT :usr IDENTIFIED BY :pwd USING :conn_opt3;
```

[Example 3] The following example is to set two connection methods in USING clause and to connect to the database server. Attempts to connect to the database server using the user name and the user password stored in usr and pwd and connection information stored in conn_opt1. In case the connection fails, it attempts again to connect to the database server by referring the user name and the user password and connection information stored in conn_opt2.

```
< Example Program : connect2.sc >
EXEC SQL BEGIN DECLARE SECTION;
char usr[10];
char pwd[10];
char conn_opt1[100];
char conn_opt2[100];
EXEC SQL END DECLARE SECTION;
strcpy(usr, "SYS");
strcpy(pwd, "MANAGER");
strcpy(conn_opt1, "DSN=192.168.11.12;CONNTYPE=1;PORT_NO=53000");
strcpy(conn_opt2, "DSN=192.168.11.22;CONNTYPE=1;PORT_NO=53000");
EXEC SQL CONNECT :usr IDENTIFIED BY :pwd USING :conn_opt1, :conn_opt2;
if (sqlca.sqlcode == SQL_SUCCESS) /* check sqlca.sqlcode */
{
 printf("Success connection to ALTIBASE server with first option\n\n");
}
else if (sqlca.sqlcode == SQL_SUCCESS_WITH_INFO)
{
 /* fail connection with first option and then success connection with second
option */
 printf("Success connection to ALTIBASE server with second option\n");
 printf("First connection error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqler-
rmc);
}
else
{
 printf("Fail connection to ALTIBASE server both first option and second 
option\n");
 printf("Error : [%d]\n", SQLCODE);
 printf("%s\n\n", sqlca.sqlerrm.sqlerrmc);
 exit(1);
}
< Example Program : Using xa connect >
EXEC SQL BEGIN DECLARE SECTION;
int mainRmid;
char usr[10];
char pwd[10];
EXEC SQL END DECLARE SECTION;
mainRmid = 1;
/* Access to server by xa_open */
xaRc = xa_open(xaOpenFormat, mainRmid, TMNOFLAGS);
printf("xa_open(%s, %d, TMNOFLAGS) = %d\n", xaOpenFormat, mainRmid, xaRc);
/* If use xa, rmid value transfer to host variable */
EXEC SQL CONNECT :usr IDENTIFIED BY :pwd ENABLE XA RMID :rmid;
```

DISCONNECT

Disconnects from the database server.

Syntax

```
EXEC SQL DISCONNECT;
```

Argument

None

Description

Disconnects from the database server, and releases all resources allocated to the connection.

Example

The following is an example of DISCONNECT statement.

```
< Example Program : connect1.sc >
EXEC SQL DISCONNECT;
```

Sample Programs

connect1.sc

See $ALTIBASE_HOME/sample/APRE/connect1.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make connect1
shell> connect1
<CONNECT 1>
----------------------------------------------------
[Connect]
----------------------------------------------------
Success connection to altibase server
----------------------------------------------------
[Disconnect]
----------------------------------------------------
Success disconnection from altibase server
```

connect2.sc

```
See $ALTIBASE_HOME/sample/APRE/connect2.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make connect2
shell> connect2
<CONNECT 2>
----------------------------------------------------
[Connect With Two ConnOpt]
----------------------------------------------------
Fail connection to altibase server both first option and second option
Error : [-327730]
Failed first connection : Client unable to establish connection
Failed second connection : Client unable to establish connection
```

Using DDL and DML in Embedded SQL Statements

Basic embedded SQL statements include DML statement such as SELECT, UPDATE, INSERT, and DELETE and DDL statement such as CREATE, DROP, and ALTER.

SELECT

Searches the records meeting the conditions in the database, and stores them in the host variable. The basic syntax is same as SELECT statement of Altibase SQL. However, to use the host variable, INTO clause is additionally needed.

Syntax

```
EXEC SQL SELECT [ ALL | DISTINCT ] <target_list>

INTO <host_var_list>

FROM <table_expression> [ WHERE … ];
```

Arguments

<target_list> : See SQL User’s Manual.

<host_var_list> : Output host variable and output indicator variable list

<table_expression> : See SQL User’s Manual.

Description

Unless the host variable is not array, only one record must be returned. In case more than one record is returned, "Returns too many rows" error message will be displayed. In this case, use CURSOR statement.

In case the host variable is the array, the number of returned records must be same as or less than the array size. In case the number of returned records is more than the array size, "Returns too many rows" error message will be displayed. In this case, increase the array size or use CURSOR statement.

Result

When the host variable is not an array	When the host variable is an array		
Number of returned records	Execution Result	Number of returned records	Execution Result

When the host variable is not an array	When the host variable is an array		
0	SQL_NO_DATA	0	SQL_NO_DATA
1	SQL_SUCCESS	If it is smaller than array size	SQL_SUCCESS
		If it is the same as array size	SQL_SUCCESS
If it is bigger than 1	SQL_ERROR	If it is bigger than array size	SQL_ERROR

When the execution result is SQL_NO_DATA, zero record will be returned. Therefore, the host variable will not have any meaning (garbage value.)

Restrictions

The input host variable must not be an array.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1;
int var2[10];
int var3[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT * INTO :var1
FROM T1 WHERE i1 = :var3;(X)
 or
EXEC SQL SELECT * INTO :var2
FROM T1 WHERE i1 = :var3;(X)
```

If the host variable in INTO clause is an array of the structure, only one output host variable must be used. In other words, it must not be used together with other host variables. If an array type of the structure is used in INTO clause, the number of structure components must be the same with the number of columns in SELECT clause.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1[10];
struct tag1 { int i3; int i4; } var2[10];
int var3;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT I1, I2 INTO :var1
FROM T1 WHERE I1 = :var3;(O)
EXEC SQL SELECT I1, I2, I3, I4
INTO :var1, :var2
FROM T1 WHERE I1 = :var3;(X)
```

According to the above limitations, when the host variable in INTO clause is an array of the varchar, only one output host variable must be used because the varchar type is a structure internally. Therefore, if the array type of varchar is used in INTO clause, there must be one column in SELECT clause.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
```

```
varchar var1[10];
int var2[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2)
VALUES (:var1, :var2);(X)
```

The second and the third limitations are due to the internal regulation that requires the structure to include all host variables and indicator variables when the host variable is an array of the structure.

In LIMIT clause of SELECT statement, only the input host variable, not the input indicator variable, can be used. Also, the datatype of the input host variable supports only the INTEGER (Not supported yet.)

Example

Shows the example of various SELECT statement.

[Example 1] In the following example, the DNO searches records with s_dno, and stores DNAME and DEP_LOCATION column in s_dname ands_dep_location host variables each.

< Example Program : select.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
char s_dname[30+1];
char s_dep_location[9+1];
EXEC SQL END DECLARE SECTION;
s_dno = 1001;
EXEC SQL SELECT DNAME, DEP_LOCATION
INTO :s_dname, :s_dep_location
FROM DEPARTMENT
WHERE DNO = :s_dno;
```

[Example 2] The following example is when the host variable of the structure type ios used. In the following example, the DNO searches records with s_dno, and stores column values in the corresponding components of s_department.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
} department;
EXEC SQL END DECLARE SECTION;
```

< Example Program : select.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
department s_department;
EXEC SQL END DECLARE SECTION;
s_dno = 1002;
```

```
EXEC SQL SELECT *
INTO :s_department
FROM DEPARTMENT
WHERE DNO = :s_dno;
```

[Example 3] The following example shows how to save CLOB column created with APRE_FILE_CREATE option in sI2FName file and INTEGER column in sI1 host variable respectively after searching for T_LOB table.

<Example Program : clobSample.sc>

```
EXEC SQL BEGIN DECLARE SECTION;
int sI1;
char sI2FName[33];
unsigned int sI2FOpt;
SQLLEN sI2Ind;
EXEC SQL END DECLARE SECTION;
strcpy(sI2FName, aOutFileName);
sI2FOpt = APRE_FILE_CREATE;

EXEC SQL SELECT * INTO :sI1, CLOB_FILE :sI2FName OPTION :sI2FOpt INDICATOR
:sI2Ind FROM T_LOB;
```

* BLOB example is in blobSample.sc and similar to CLOB's.

INSERT

Inserts new records in the table.

Syntax

See SQL User's Manual.

Argument

None

Description

In VALUES clause, the host variable and the indicator variable can be used.

Example

Shows various examples of INSERT statement.

[Example 1] The following example is to insert new records in GOODS table.

< Example Program : insert.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
char s_gno[10+1];
char s_gname[20+1];
char s_goods_location[9+1];
int s_stock;
double s_price;
```

```
EXEC SQL END DECLARE SECTION;
strcpy(s_gno, "F111100002");
strcpy(s_gname, "XX-101");
strcpy(s_goods_location, "FD0003");
s_stock = 5000;
s_price = 9980.21;
EXEC SQL INSERT INTO GOODS
VALUES (:s_gno, :s_gname, :s_goods_location,
:s_stock, :s_price);
```

[Example 2] The following example is to insert new records in GOODS table using the structure-type host variable.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct goods
{
 char gno[10+1];
 char gname[20+1];
 char goods_location[9+1];
 int stock;
 double price;
} goods;
EXEC SQL END DECLARE SECTION;
```

< Example Program : insert.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
goods s_goods;
EXEC SQL END DECLARE SECTION;
strcpy(s_goods.gno, "F111100003");
strcpy(s_goods.gname, "XX-102");
strcpy(s_goods.goods_location, "AD0003");
s_goods.stock = 6000;
s_goods.price = 10200.96;
EXEC SQL INSERT INTO GOODS VALUES (:s_goods);
```

[Example 3] The following example shows how to insert new record into T_LOB table after reading BLOB data in file with APRE_FILE_READ.

<Example Program : blobSample.sc>

```
EXEC SQL BEGIN DECLARE SECTION;
int sI1;
char sI2FName[32];
unsigned int sI2FOpt;
SQLLEN sI2Ind;
EXEC SQL END DECLARE SECTION;
sI1 = 1;
strcpy(sI2FName,aInputFileName);
sI2FOpt = APRE_FILE_READ;
EXEC SQL INSERT INTO T_LOB VALUES(:sI1, BLOB_FILE :sI2FName OPTION :sI2FOpt
INDICATOR :sI2Ind);
```

UPDATE

Search the records meeting the conditions, and change the explicit column values.

Syntax

See SQL User's Manual.

Argument

None

Description

In SET clause and WHERE clause, the host variable and the indicator variable can be used.

Restrictions

- The array type of the structure must not be used. The reason is that it is not possible to define array elements in the embedded SQL statement.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; int i3; } var1[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL UPDATE T1
SET I1 = :var1[0].i1, I2 = :var1[0].i2
WHERE I1 = :var1[0].i3;(X)
```

- The array type must not be used together with a non-array type. For example, if the host variable in SET clause is the array type, the host variable in WHERE clause must be the array type.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1[10];
int var2[10];
int var3;
EXEC SQL END DECLARE SECTION;
EXEC SQL UPDATE T1
SET I1 = :var1, I2 = :var2
WHERE I1 = :var3; (X)
```

Example

Shows various examples of UPDATE statement.

[Example 1] In the following example, DNO and EMP_JOB columns are converted into s_dno and s_emp_job.arr each.

< Example Program : update.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
short s_dno;
varchar s_emp_job[15+1];
EXEC SQL END DECLARE SECTION;
s_eno = 2;
```

```
s_dno = 1001;
strcpy(s_emp_job.arr, "ENGINEER");
s_emp_job.len = strlen(s_emp_job.arr);
EXEC SQL UPDATE EMPLOYEE
SET DNO = :s_dno,
 EMP_JOB = :s_emp_job
WHERE ENO = :s_eno;
```

[Example 2] The following example is when the structure-type host variable is used. In the following example, DNO, EMP_JOB, and JOIN_DATE columns are converted into s_employee.s_dno, s_employee.s_emp_job.arr and SYSDATE respectively.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct employee
{
 int eno;
 char ename[20+1];
 varchar emp_job[15+1];
 char emp_tel[15+1];
 short dno;
 double salary;
 char sex;
 char birth[4+1];
 char join_date[19+1];
 char status[1+1];
} employee;
EXEC SQL END DECLARE SECTION;
```

< Example Program : update.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
employee s_employee;
EXEC SQL END DECLARE SECTION;
s_eno = 20;
s_employee.dno = 2001;
strcpy(s_employee.emp_job.arr, "TESTER");
s_employee.emp_job.len = strlen(s_employee.emp_job.arr);
EXEC SQL UPDATE EMPLOYEE
SET DNO = :s_employee.dno,
 EMP_JOB = :s_employee.emp_job,
 JOIN_DATE = SYSDATE
 WHERE ENO = :s_eno;
```

DELETE

Delete the records meeting conditions from the corresponding table.

Syntax

See SQL User's Manual.

Argument

None

Description

In WHERE clause, the host variable and the indicator variable can be used.

Examples

The following shows how to delete the records meeting the conditions from the EMPLOYEE table.

```
< Example Program : delete.sc >
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
short s_dno;
EXEC SQL END DECLARE SECTION;
s_eno = 5;
s_dno = 1000;
EXEC SQL DELETE FROM EMPLOYEE
 WHERE ENO > :s_eno AND
DNO > :s_dno AND
EMP_JOB LIKE 'P%';
```

Sample Programs

select.sc

```
See $ALTIBASE_HOME/sample/APRE/select.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make select
shell> ./select
<SELECT>
----------------------------------------------------
[Scalar Host Variables]
----------------------------------------------------
DNO DNAME DEP_LOCATION
----------------------------------------------------
1001 RESEARCH DEVELOPMENT DEPT 1 New York
----------------------------------------------------
[Structure Host Variables]
----------------------------------------------------
DNO DNAME DEP_LOCATION MGR_NO
----------------------------------------------------
1002 RESEARCH DEVELOPMENT DEPT 2 Sydney 13
----------------------------------------------------
[Error Case : Scalar Host Variables]
----------------------------------------------------
Error : [-594092] Returns too many rows
```

insert.sc

```
See $ALTIBASE_HOME/sample/APRE/insert.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make insert
shell> ./insert
<INSERT>
-----------------------------------------------------
[Scalar Host Variables]
-----------------------------------------------------
1 rows inserted
-----------------------------------------------------
[Structure Host Variables]
-----------------------------------------------------
1 rows inserted
```

update.sc

```
See $ALTIBASE_HOME/sample/APRE/update.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make update
shell> ./update
<UPDATE>
-----------------------------------------------------
[Scalar Host Variables]
-----------------------------------------------------
1 rows updated
-----------------------------------------------------
[Structure Host Variables]
-----------------------------------------------------
1 rows updated
```

delete.sc

```
See $ALTIBASE_HOME/sample/APRE/delete.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make delete
shell> ./delete
<DELETE>
-----------------------------------------------------
[Scalar Host Variables]
-----------------------------------------------------
7 rows deleted
```

Using Other Embedded SQL Statements

Includes work control statement, system control statement, transaction statement, INCLUDE statement, and THREADS statements.

AUTOCOMMIT

Changes the AUTOCOMMIT mode of the current session.

Syntax

```
EXEC SQL AUTOCOMMIT { ON | OFF };
```

Argument

None

Example

The following example is to change the AUTOCOMMIT mode.

EXEC SQL AUTOCOMMIT ON; - To change into AUTOCOMMIT mode

EXEC SQL AUTOCOMMIT OFF;

- To change into Non- AUTOCOMMIT mode.

COMMIT

Successfully terminates the current transaction. The transaction result will be stored in the database permanently.

Syntax

```
EXEC SQL COMMIT;
```

Argument

None

Description

When the current session is AUTOCOMMIT, an error will occur.

Example

The following is an example of COMMIT statement.

EXEC SQL COMMIT;

SAVEPOINT

The save point is to temporarily store the current transactions. This embedded SQL statement specifies the save point ton indicate the roll-back point in the transaction.

Syntax

```
EXEC SQL SAVEPOINT <savepoint_name>;
```

Argument

<savepoint_name> : Name of the storing point

Description

When the current session is AUTOCOMMIT, an error will occur.

Example

The following is an example of SAVEPOINT statement.

EXEC SQL SAVEPOINT sp;

ROLLBACK

Returns the statement to the state before the DDL statement or COMMIT statement was executed, and undoes the transaction result.

Syntax

```
EXEC SQL ROLLBACK

[ TO SAVEPOINT <savepoint_name> ];
```

Argument

<savepoint_name> : Name of the storing point

Description

When the current session is AUTOCOMMIT, an error will occur.

When the save point is defined, the transactions from the present to the save point will be cancelled.

Example

The following is an example or ROLLBACK statement.

EXEC SQL ROLLBACK;

or

EXEC SQL ROLLBACK TO SAVEPOINT sp;

BATCH

Changes the connection properties to activate or stop the batch processing.

Syntax

```
EXEC SQL BATCH { ON | OFF };
```

Argument

None

Description

When the batch processing mode is activated, the execution of the embedded SQL statement (transmission to the server) will be delayed until SELECT statement or COMMIT is executed. This is possible because the data can be read only from uncommitted transactions such as INSERT, UPDATE, or DELETE statement.

When INSERT, UPDATE, and DELETE statements are often used, activating the batch processing mode can improve the performance.

Example

The following is an example of the BATCH processing.

```
EXEC SQL BATCH ON;- to activate the batch processing mode.
EXEC SQL BATCH OFF; - Not to activate the batch processing mode.
```

FREE

Releases all resources allocated upon the connection with the database server and execution of the embedded SQL statement.

Syntax

```
EXEC SQL FREE;
```

Argument

None

Description

In case the server is disconnected during execution of the embedded SQL statement, execute FREE statement before establishing connection again. At this time, the database server must not be running. If the database server is running, execute DISCONNECT statement instead of FREE statement.

Example

The following is an example of FREE statement.

< Example Program : free.sc >

```
EXEC SQL FREE;
```

INCLUDE

Specifies the header file to be used for precompiling.

Syntax

```
EXEC SQL INCLUDE <filename>;
```

This syntax can be used for the file (.sc) to be precompiled and the header file (.h) used for precompiling. However, this cannot be used for the header file run by the #include command.

Argument

<filename> : The name of the header file to be used for precompiling

Description

Definition of the datatype for the host variable and the host variable is important information that C/C++ precompiler must know for precompiling. Therefore, the header file with declaration of the host variable of with the definition of the host variable type must be included by INCLUDE statement.

Restrictions

The header file cannot be mutually referred to. In other words, myheader1.h must not refer to myheader2.h and myheader2.h must not refer to myheader1.h.

```
Example) <myheader1.h>
EXEC SQL INCLUDE myheader2.h;
...
<myheader2.h>
EXEC SQL INCLUDE myheader1.h;(X)
```

Example

The following example is to set the header file to use for precompiling using Syntax No. 1.

< Example Program : insert.sc >

```
EXEC SQL INCLUDE hostvar.h;
```

Sample Programs

free.sc

```
See $ALTIBASE_HOME/sample/APRE/free.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make free
shell> ./free
<FREE>
----------------------------------------------------
[Connect]
----------------------------------------------------
Success connection to altibase server
----------------------------------------------------
[Free]
----------------------------------------------------
Error : [-331796] Function sequence error
----------------------------------------------------
[Reconnect]
----------------------------------------------------
Error : [-589826] Already connected
```

OPTION Statements

Sets the options provided by C/C++ precompiler using OPTION statement.

INCLUDE

To set the location of the header file used for precompiling, the embedded SQL statement provides various methods. One of them is Include OPTION statement.

Syntax

```
EXEC SQL OPTION (INCLUDE = <pathname>);
```

Argument

<pathname> : The location of the header file to be used for precompiling

Description

Specifies the location of the header file to be used for precompiling.

Each location must be separated by comma. This OPTION statement must be declared before INCLUDE statement.

Example

The following example is to set the location of hostvar.h (./include directory) using INCLUDE OPTION statement and to include hostvar.h.

< Example Program : insert.sc >

```
EXEC SQL OPTION (INCLUDE=./include);
EXEC SQL INCLUDE hostvar.h;
```

THREADS

The embedded SQL statement supports the multi-threaded program. This OPTION statement provides a background for the precompiler to judge whether the file to be precompiled is a multi-threaded program or not.

Syntax

```
EXEC SQL OPTION (THREADS = { TRUE | FALSE });
```

Argument

None

Description

TRUE : When the file to be precompiled is a multi-threaded program

FALSE: When the file to be precompiled is not a multi-threaded program

The default value of THREADS OPTION is FALSE. If the file to be precompiled is a multi-threaded program, THREADS OPTION must be TRUE. In case -mt option is used in the command line when a multi-threaded program is precompiled, OPTION statement can be omitted.

Example

The following example is to set TRUE value for THREADS OPTION using OPTION statement when the file to be precompiled is a multi-threaded program.

```
< Example Program : mt1.sc >

EXEC SQL OPTION (THREADS=TRUE);
```

8 Handling Runtime Errors

Overview

The application must be able to handle the execution time error. The embedded SQL statement must support the variables such as SQLCODE and SQLSTATE and WHENEVER statement to provide the execution time error processing method for the programmer.

Return Values

The execution result of the embedded SQL statement is stored in .sqlcode, and the value is as follows:

SQL_SUCCESS

Upon successful execution of the embedded SQL statement

SQL_SUCCESS_WITH_INFO

When an exception is found after the embedded SQL statement is executed

SQL_NO_DATA

When there is no record returned after SELECT or FETCH statement is executed:

SQL_ERROR

When an error occurs during the embedded SQL statement is executed

Using SQLCA Struncture

sqlca is declared upon precompiling and an instance of ulpSqlca structure. ulpSqlca is a structure to store the execution result of the embedded SQL statement, and is defined in ulpLibInterface.h file. The program developer can refer to the execution result of the embedded SQL statement using sqlca variable in the application.

Data Structure Definition

```
typedef struct ulpSqlca
{
 char sqlcaid[8]; /* not used */
 int sqlcabc; /* not used */
 int sqlcode;
 struct
 {
 short sqlerrml;
 char sqlerrmc[2048];
 }sqlerrm;
 char sqlerrp[8]; /* not used */
 int sqlerrd[6];
 char sqlwarn[8]; /* not used */
 char sqlext[8]; /* not used */
}apre_sqlca;
```

Elements

ulpSqlca structure includes various components. Some components are reserved for the future use, and such components are not described here.

The meaning of each component is as follows:

sqlcode

The execution result of the embedded SQL statement is stored. The execution result of the embedded SQL statement is as follows:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA
- SQL_ERROR

sqlerrm.sqlerrmc

The error message is stored. The maximum length of the error message to save is 2048 bytes.

sqlerrm.sqlerrml

The length of the returned error message is stored.

sqlerrd[2]

The number of records affected by INSERT, UPDATE, and DELETE operations are stored.

The number of returned records when the output host variable is array upon execution of SELECT statement or FETCH statement is stored. In this case, the number of returned records is not accumulated. Instead, the number of currently fetched records is stored. Therefore, this value must not be larger than the array size.

sqlerrd[3]

When an array-type input host variable is used, this value can be referred to after the embedded SQL statement is executed. In this variable, the successful processing count is stored. Therefore, this value must not be larger than the array size. For example, if an input host variable of which array size is 3 executed UPDATE, "2" will be stored in this variable in case the first attempt is success, the second attempt is failure, and the third attempt is success. At this time, the number of updated records is stored in sqlca.sqlerrd[2], which means a higher value than 2 can be stored.

Precautions

- After execution of every embedded SQL statement, sqlca.sqlcode must be checked for accurate error handling.
- In case the size of the output variable is same as or smaller than the corresponding column size in SELECT statement, the data will be cut to be stored in the host variable. At this time, sqlca.sqlcode will be SQL_SUCCESS_WITH_INFO.
- In case no record is affected by UPDATE or DELETE operation, sqlca.sqlcode will be SQL_NO_DATE. To check the number of records affected by UPDATE or DELETE operation, see sqlca.sqlerrd[2].

Using SQLCODE

In SQLCODE, the execution result of the embedded SQL statement will be stored. If the execution result is SQL_ERROR, the error code will be stored.

Data structure definition

```
int SQLCODE
```

Description

0 : Upon successful execution of the embedded SQL statement ;sqlca.sqlcode is SQL_SUCCESS

1 : When an exception is detected after execution of the embedded SQL statement ;sqlca.sqlcode is SQL_SUCCESS_WITH_INFO:

100 : When there is no record returned after SELECT or FETCH statement is executed; sqlca.sqlcode is SQL_NO_DATA:

-1 : When there is no corresponding error code for the error occurring during execution of an embedded SQL statement: At this time, sqlca.sqlcode is SQL_ERROR.

- Other negative values : When an error occurs upon execution of the embedded SQL statement

Error Code

Depending on the location of the error occurrence, the error codes are divided into the databa server errors and the embedded SQL statement errors.

Embedded SQL statement error

An error occurring in the embedded SQL statement curing execution. In this case, the C/C++ pre-compiler will return the error.

-589825 : When an error occurs when allocating a memory for connection to the database server.

-589826 : In case a connection with the same name already exists

-589841 : In case the name of the connection exceeds 50 characters

-589857 : When a cursor-processing SQL statement is executed without an undeclared cursor name.

-589858 : When a dynamic SQL statement is executed with an unprepared SQL statement identifier

-593921 : One or more than the number of arrays are returned in SELECT…INTO statement.

-598017 : When the processing count specified in FOR clause smaller than 1.

-598018 : When the processing count specified in FOR is larger than the array size.

Database server error

An error occurring in the database server during execution. In this case, the database server will return the error code. For more information about each error code, see the Error Message Reference.

Precautions

In the SQCODE, the error code is a negative value. However, in Error Message Reference , the error code is a positive hexadecimal value. Therefore, when referring to Error Message Reference, convert the absolute value of the SQLCODE into a hexadecimal data.

Using SQLSTATE

The status code is stored in SQLSTATE. Through this status code, the user can check which kind of error or exception has occurred. When the execution result of the embedded SQL statement is SQL_ERROR or SQL_SUCCESS_WITH_INFO, SQLSTATE can be referred to.

Definition of Data Structure

```
Char SQLSTATE[6]
```

Status Code

00000 – Upon successful execution of the embedded SQL statement; When the size of the host variable is same as or smaller than the corresponding column size when the host variable is the character type. At this time, the returned data is cut to be stored in the host variable.

07006 – When the host variable type is not compatible with the corresponding column type

07009 – When the number of the columns is higher than the number of corresponding host variables

08001 – When the database server is not started up

08S01 – When the database server is disconnected

22002 – When NULL data are returned while no indicator variable is specified

HY000 – General error

HY001 – When an error occurs upon allocation of the memory

HY009 – When the host variable and the indicator variable are null points

HY010 – When an unopened cursor is fetched

HY090 – When the indicator variable is a negative value

WHENEVER Statement

The embedded SQL statement supports WHENEVER statement to handle the execution time errors.

Syntax

```
EXEC SQL WHENEVER <condition> <action>;
```

Argument

<condition> : Execution result of the embedded SQL statement

<action> : Processing method depending on the execution result of the embedded SQL statement

Conditions

The following conditions can be set in WHENEVER statement:

SQLERROR

In case of an error upon execution of the embedded SQL statement In other words, this is when sqlca.sqlcode is SQL_ERROR.

NOT FOUND

When there is no record returned after SELECT or FETCH statement is executed: In other words, this is when sqlca.sqlcode is SQL_NO_DATA.

Processing Controls

In case the execution result of the embedded SQL statement matches with the conditions specified in WHENEVER statement, the processing will be made as specified.

The processing methods that can be used in WHENEVER statement are as follows:

CONTINUE

Continue.

DO BREAK

Exits the current repetition, and continues. This has same effects as using "break;" command in the repetition. This can be specified in "DO BREAK" repetition. After the repetition ends, WHENEVER statement becomes invalid.

DO CONTINUE

Goes to the beginning of the current repetition, and continues. This has the same effects as using "Continue;" command in the repetition. "DO CONTINUE" can be specified only within the repetition. After the repetition, WHENEVER statement becomes invalid.

DO function_name

This calls the function specified as function_name.

GOTO label_name

Goes to label_name, and continues.

STOP

Disconnects from the database server, and terminates the current program.

Description

The application scope of WHENEVER statement is different from the program flow and is valid only within the current file.

WHENEVER statement must be declared before the embedded SQL statement to be applied.

Once WHENEVER statement is declared, the execution results of all embedded SQL statement in the current scope and the lower scope are affected. In other words, when the execution result of the embedded SQL statement matches with the condition specified by WHENEVER statement, the corresponding action will be executed.

WHENEVER statement is independent of two conditions - "SQLERROR" and "NOT FOUND."

Out of the scope where WHENEVER statement is declared, the WHENEVER statement is not valid. The embedded SQL statement will be affected by the WHENEVER statement of the current scope or the upper scope.

If another WHENEVER statement is declared in a scope where a WHENEVER statement already exists, the previous statement will lose it effect and the newly declared WHENEVER statement will be applied.

In case two WHENEVER statements with the same conditions are declared, the nearer WHENEVER statement will take effect.

WHENEVER statement is independent of the connection. In other word, declaration of WHENEVER statement in a file with one or more connections, all embedded SQL statements within the corresponding range will be affected regardless of the connection.

In case WHENEVER statement is globally declared, all embedded SQL statements of the current file will be affected.

Sample Programs

runtime_error_check.sc

```
See $ALTIBASE_HOME/sample/APRE/runtime_error_check.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make runtime_error_check
shell> ./runtime_error_check
<RUNTIME ERROR CHECK>
-----------------------------------------------------
[SQL_SUCCESS]
-----------------------------------------------------
sqlca.sqlcode = 0
------------------------------------------------------------
[SQL_SUCCESS_WITH_INFO With SQLSTATE=01004]
-----------------------------------------------------
sqlca.sqlcode = 1
sqlca.sqlerrm.sqlerrmc = String data right truncated.
SQLSTATE = 01004
SQLCODE = 1
-----------------------------------------------------
[SQL_ERROR With SQLSTATE=22002]
-----------------------------------------------------
sqlca.sqlcode = -1
sqlca.sqlerrm.sqlerrmc = Indicator variable required but not supplied.
SQLSTATE = 22002
SQLCODE = -331841
-----------------------------------------------------
[SQL_NO_DATA With SELECT]
-----------------------------------------------------
sqlca.sqlcode = 100
sqlca.sqlerrm.sqlerrmc = Not found data
SQLSTATE = 02000
SQLCODE = 100
-----------------------------------------------------
[SQL_NO_DATA With FETCH]
------------------------------------------------------------sqlca.sqlcode = 
100
sqlca.sqlerrm.sqlerrmc = Not found data
SQLSTATE = 02000
SQLCODE = 100
2 rows fetched
-----------------------------------------------------
[SQL_ERROR]
-----------------------------------------------------
sqlca.sqlcode = -1
sqlca.sqlerrm.sqlerrmc = The row already exists in a unique index.
SQLSTATE = 23000
SQLCODE = -69720
-----------------------------------------------------
[SQL_ERROR With SQLSTATE=HY010]
-----------------------------------------------------
sqlca.sqlcode = -1
sqlca.sqlerrm.sqlerrmc = Function sequence error.
SQLSTATE = HY010
SQLCODE = -331796
```

```
---------------------------------------------------
[sqlca.sqlerrd[2]]
---------------------------------------------------
sqlca.sqlcode = 0
sqlca.sqlerrd[2] = 12
---------------------------------------------------
sqlca.sqlerrd[3] With Array In-Binding]
---------------------------------------------------
sqlca.sqlcode = 0
sqlca.sqlerrd[2] = 12
sqlca.sqlerrd[3] = 3
```

whenever1.sc

```
See $ALTIBASE_HOME/sample/APRE/whenever1.sc
```

whenever2.sc

```
See $ALTIBASE_HOME/sample/APRE/whenever2.sc
```

Execution Result

```
shell> is –f schema/schema.sql
shell> make whenever
shell> ./whenever
<WHENEVER>
Success connection
---------------------------------------------------
DNO DNAME DEP_LOCATION MGR_NO
---------------------------------------------------
1001 PAPER TEAM New York 16
1002 RESEARCH DEVELOPMENT DEPT 2 Sydney 13
1003 SOLUTION DEVELOPMENT DEPT Japan 14
2001 QUALITY ASSURANCE DEPT Seoul 17
3001 CUSTOMER SUPPORT DEPT London 4
3002 PRESALES DEPT Peking 5
4001 MARKETING DEPT Seoul 8
4002 BUSINESS DEPT LA 7
```

9 Cursor Statements

Overview

If multiple records are returned as a result of an inquiry, a cursor can be declared.

Includes various embedded SQL statements that declare and manipulate the cursor.

Description

Cannot declare more than one cursor name in one program. In this case, only the latest cursor declaration will be valid. Therefore, when executing Cursor OPEN statement, Cursor FETCH statement, Cursor CLOSE statement, and Cursor CLOSE RELEASE statement, refer to the latest cursor declaration.

In case Cursor OPEN statement, Cursor FETCH statement, Cursor CLOSE statement, or Cursor CLOSE RELEASE statement is executed without an undeclared cursor name, "Not defined" error message will be displayed.

Execution Steps of the Cursor Statements

The execution order of the cursor-processing SQL statement is as follows:

Cursor declaration

Cursor OPEN statement

Cursor FETCH statement

To bring all records meeting the conditions, repeatedly execute Cursor FETCH statement till the execution result is SQL_NO_DATA.

Cursor CLOSE statement or Cursor CLOSE RELEASE statement

Using Cursor Statements

Cursor-processing SQL statements are divided into Cursor Declaration statement, Cursor OPEN statement, Cursor FETCH statement, Cursor CLOSE statement, and Cursor CLOSE RELEASE statement depending on the cursor processing and manipulation methods. The following describes each cursor-processing SQL statement.

DECLARE CURSOR

Declares the cursor.

Syntax

EXEC SQL DECLARE <cursor name> CURSOR FOR

<cursor specification>;

Argument

<cursor name> : Cursor name Up to maximum 50 characters

<cursor specification> : Select statement in Altibase SQL. See SQL User's Manual.

Description

Cursor Declaration statement must be executed first among cursor-processing SQL statements. When a cursor-processing SQL statement is executed with an undeclared cursor name, "Not defined" error message will be displayed.

Cursor Declaration statement only prepares for execution of the SQL statement such as syntax checking, semantic checking, optimizing, plan tree creation. By preparing for the execution of the SQL statement, the user can use the preparation (prepare-DECLARE CURSOR) for multiple executions (execute-OPEN CURSOR.)

Restrictions

The restrictions in SELECT statement is applied.

Example

The following example is to declare a cursor that will search all records in the DEPARTMENT table.

```
< Example Program : cursor1.sc >
EXEC SQL DECLARE DEPT_CUR CURSOR FOR
 SELECT *
 FROM DEPARTMENT;
```

OPEN

Opens the cursor.

Syntax

EXEC SQL OPEN <cursor name>;

Argument

<cursor name> : Cursor name

Description

Cursor OPEN statement executes the SQL statement in the Cursor Declaration statement.

The SQL statement already prepared in Cursor Declaration statement is executed. When the SQL statement is executed, the database server searches records meeting the conditions in the table.

Example

The following example is to OPEN DEPT_CUR.

```
< Example Program : cursor1.sc >
EXEC SQL OPEN DEPT_CUR;
```

FETCH

Reads the column values from the opened cursor and stores them in the corresponding host variables.

Syntax

```
EXEC SQL FETCH <cursor name>
INTO <host_var_list>;
```

Argument

<cursor name> : Cursor name

<host_var_list> : Output host variable and output indicator variable list

Description

Cursor FETCH statement stores the returned columns in the corresponding host variables.

Execution Result

The following describes when the execution result of Cursor FETCH statement is SQL_SUCCESS and when it is SQL_NO_DATA.

When the result is SQL_SUCCESS

The current fetching result has been successfully stored in the host variable, and the database server still has data to return.

In general, fetching continues when the result is SQL_SUCCESS.

When the result is SQL_NO_DATA

As there is no fetching result and no data is stored in the host variable, the host variable does not have any meaning (garbage value.) The database server returned all records meeting the conditions, or there is no records meeting the conditions.

Example

The following is an example of fetching DEPT_CUR. The returned columns are stored in the components of s_department. Using s_dept_ind indicator variable, the user can check whether the returned column is NULL or not. Fetches and brings records meeting the conditions till SQL_NO_DATA is returned in While loop.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
} department;
typedef struct dept_ind
{
 int dno;
 int dname;
 int dep_location;
 int mgr_no;
} dept_ind;
EXEC SQL END DECLARE SECTION;
```

< Example Program : cursor1.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
/* declare host variables */
department s_department;
/* structure indicator variables */
dept_ind s_dept_ind;
EXEC SQL END DECLARE SECTION;
while(1)
{
EXEC SQL FETCH DEPT_CUR
 INTO :s_department :s_dept_ind;
 if (sqlca.sqlcode == SQL_SUCCESS)
{
 printf("%d %s %s %d\n",
 s_department.dno, s_department.dname,
 s_department.dep_location,
s_department.mgr_no);
 }
 else if (sqlca.sqlcode == SQL_NO_DATA)
 {
 break;
 }
 else
 {
 printf("Error : [%d] %s\n", SQLCODE,
sqlca.sqlerrm.sqlerrmc);
 break;
 }
}
```

CLOSE

Close the cursor.

Syntax

EXEC SQL CLOSE <cursor name>;

Argument

<cursor name> : Cursor name

Description

As long as the database server has data to return, Cursor CLOSE statement deletes the result (if fetching is not fully made.) In other words, once Cursor CLOSE statement is executed, Cursor FETCH statement cannot be executed with the same cursor name. If the user intends to fetch the same cursor name, rename the cursor after opening.

If there is no result to return (after fetching is fully made), execution of Cursor CLOSE statement will yield no result. Therefore, after fetching is fully made, Cursor CLOSE statement can be omitted.

Cursor CLOSE statement does not release the resources allocated to the cursor and stores the preparation of the SQL statement made by Cursor Declaration statement. When using the same cursor name after executing Cursor CLOSE statement, the user can directly execute Cursor OPEN statement omitting Cursor Declaration statement.

Example

The following is an example of closing DEPT_CUR.

```
< Example Program : cursor1.sc >
EXEC SQL CLOSE DEPT_CUR;
```

CLOSE RELEASE

Closes the cursor and releases the resources allocated to the cursor.

Syntax

EXEC SQL CLOSE RELEASE <cursor name>;

Argument

<cursor name> : Cursor name

Description

Cursor CLOSE RELEASE statement releases resources allocated to the cursor and deletes the SQL statement preparation made by Cursor Declaration statement. In case there is a result to be returned, the result will be deleted. When using the same cursor name after executing Cursor CLOSE RELEASE statement, execute Cursor Declaration statement and Cursor OPEN statement in order. In other words, after executing Cursor CLOSE RELEASE statement, the user cannot execute Cursor OPEN statement with the same cursor name.

Example

The following is an example of close-releasing EMP_CUR. At this time, the SQL statement preparation made by the declarative statement of EMP_CUR will be deleted, and the resources allocated to EMP_CUR will be cancelled.

```
< Example Program : cursor2.sc >
EXEC SQL CLOSE RELEASE EMP_CUR;
```

Using the Same Cursor Name

The following describes how to use the same cursor name again: The following describes the order of repeatedly using the same cursor name and things to note when using the same cursor name:

Relations between Cursor Statements

The following describes the order of executing cursor-processing statements when using the same cursor name:

Cursor declaration

Cursor Declaration statement must be executed after Cursor CLOSE statement and Cursor CLOSE RELEASE statement.

Cursor OPEN statement

Cursor OPEN statement must be executed after Cursor FETCH statement or Cursor CLOSE statement in case the fetching is fully made.

Cursor FETCH statement

Cursor FETCH statement must be executed after Cursor OPEN statement, or in case the fetching result is SQL_SUCCESS, it must be executed after Cursor FETCH statement.

Cursor CLOSE statement

Cursor CLOSE statement can be executed after Cursor Declaration statement, Cursor OPEN statement, and Cursor FETCH statement (whether the result is SQL_SUCCESS or SQL_NO_DATA.)

Cursor CLOSE RELEASE statement

Cursor CLOSE RELEASE statement must be executed after Cursor Declaration statement, Cursor OPEN statement, Cursor FETCH statement (whether the result is SQL_SUCCESS or SQL_NO_DATA), and Cursor CLOSE statement.

Cursor SQL Statements and the Host Variables

The following describes how to use the cursor-processing SQL statement for two cases when the input variable is global and when it is local in Cursor Declaration statement:

If the host variable in Cursor Declaration statement is global, Cursor OPEN statement can be executed after Cursor CLOSE statement when the same cursor name is used again.

In case the host variable in Cursor Declaration statement Is local, Cursor Declaration statement must be executed after Cursor CLOSE statement when using the same cursor name. The pointers of the host variables use in Cursor Declaration statement are internally stored upon execution of Cursor Declaration statement, and the pointers of the host variables stored during execution of Cursor OPEN statement are used. If these host variables are local, the pointers may be changed upon execution of Cursor OPEN statement, which mean Cursor Declaration statement must be executed every time to inform the precompiler of the pointers.

CLOSE and CLOSE RELEASE

The following describes the difference between Cursor CLOSE statement and Cursor CLOSE RELEASE statement:

To use the same cursor name after Cursor CLOSE RELEASE statement, execute Cursor Declaration statement. When Cursor CLOSE RELEASE statement is executed, corresponding information and resources related to the cursor are cancelled. Therefore, the user must execute Cursor Declaration statement to allocate necessary resources to this cursor and make preparation for the execution of the SQL statement. When using the cursor again, the user must execute Cursor CLOSE statement instead of Cursor CLOSE RELEASE statement in most cases.

After fetching is complete (when the result of Cursor FETCH statement is SQL_NO_DATA), the user can execute either Cursor CLOSE statement or Cursor CLOSE RELEASE statement. To use the cursor again, execute Cursor CLOSE statement. Otherwise, execute Cursor CLOSE RELEASE statement. After Cursor CLOSE statement, Cursor CLOSE RELEASE statement can be executed. However, closing this cursor twice is waste of time.

To use the cursor again, execute CLOSE statement. Otherwise, execute CLOSE RELEASE statement. In most cases, the cursor is used again. Therefore, Cursor CLOSE RELEASE statement is rarely used. If the user selects CLOSE RELEASE -> Cursor Declaration statement -> Cursor OPEN statement to use the cursor again, the performance will be compromised.

Sample Programs

cursor1.sc

See $ALTIBASE_HOME/sample/APRE/curosor1.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make cursor1
shell> ./cursor1
<CURSOR 1>
----------------------------------------------------
[Declare Cursor]
----------------------------------------------------
Success declare cursor
----------------------------------------------------
[Open Cursor]
----------------------------------------------------
Success open cursor
----------------------------------------------------
[Fetch Cursor]
----------------------------------------------------
DNO DNAME DEP_LOCATION MGR_NO
----------------------------------------------------
1001 RESEARCH DEVELOPMENT DEPT 1 New York 16
1002 RESEARCH DEVELOPMENT DEPT 2 Sydney 13
1003 SOLUTION DEVELOPMENT DEPT Japan 14
2001 QUALITY ASSURANCE DEPT Seoul 17
3001 CUSTOMER SUPPORT DEPT London 4
3002 PRESALES DEPT Peking 5
4001 MARKETING DEPT Seoul 8
4002 BUSINESS DEPT LA 7
----------------------------------------------------
[Close Cursor]
----------------------------------------------------
Success close cursor
```

cursor2.sc

See $ALTIBASE_HOME/sample/APRE/curosor2.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make cursor2
shell> ./cursor2
<CURSOR 2>
----------------------------------------------------
[Declare Cursor]
----------------------------------------------------
Success declare cursor
----------------------------------------------------
[Open Cursor]
----------------------------------------------------
```

```
Success open cursor
----------------------------------------------------
[Fetch Cursor]
----------------------------------------------------
ENO DNO SALARY
----------------------------------------------------
2 -1 1500000.00
3 1001 2000000.00
4 3001 1800000.00
5 3002 2500000.00
6 1002 1700000.00
7 4002 500000.00
9 4001 1200000.00
10 1003 4000000.00
11 1003 2750000.00
12 4002 1890000.00
13 1002 980000.00
14 1003 2003000.00
15 1003 1000000.00
16 1001 2300000.00
17 2001 1400000.00
18 4001 1900000.00
19 4002 1800000.00
----------------------------------------------------
[Close Release Cursor]
----------------------------------------------------
Success close release cursor
```

10 Using Arrays in SQL Statements

Overview

You can declare an array-type host variable and use it. It may significantly improve your application performance and reduce network traffic.

Definition and Declaration

An array host variable refers to the host variable that has been declared as single-dimensional or two-dimensional array for the host variable datatypes.

For the Character type and the varchar type, two-dimensional array can be declared, and for others, one-dimensional array can be declared. As an exception, char* type cannot be declared as an array.

Example

Shows various examples of declaring array host variables.

[Example 1] The following is an example of declaring the character-type and the numeric-type array as host variables.

< Example Program : arrays1.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
char a_gno[3][10+1];
char a_gname[3][20+1];
char a_goods_location[3][9+1];
int a_stock[3];
double a_price[3];
EXEC SQL END DECLARE SECTION;
```

[Example 2] The following is an example of declaring an array type of the structure as a host variable.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct goods
{
 char gno[10+1];
 char gname[20+1];
 char goods_location[9+1];
 int stock;
 double price;
} goods;
EXEC SQL END DECLARE SECTION;
```

< Example Program : arrays1.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
goods a_goods[3];
EXEC SQL END DECLARE SECTION;
```

[Example 3] The following is an example of declaring the structure type of which component is an array as a host variable.

```
< Example Program : cursor1.sc >
EXEC SQL BEGIN DECLARE SECTION;
struct
```

```
{
 char gno[3][10+1];
 char gname[3][20+1];
 char goods_location[3][9+1];
 int stock[3];
 double price[3];
} a_goods2;
EXEC SQL END DECLARE SECTION;
```

Advantages

Using an array-type host variable may increase the performance.

The following describes performance improvement:

The following is a comparison of using the array-type host variable and a non-array-type host variable in INSERT statement: In case an array-type host variable is used when the array size is 1000, 1000 records will be inserted for every execution of INSERT statement. In case of a non-array-type host variable, the user must execute INSERT statement 1000 times to insert 1000 records. At this time, the communication with the data server is made 1000 times. Therefore, using an array-type host variable greatly reduces the traffic compared to using a non-array-type host variable.

The following compares using an array-type host variable with using a non-array-type host variable in FETCH statement: In case an array-type host variable is used when the array size is 1000, 1000 records will be stored from array no. 0 in order for every execution of FETCH statement. In case of a non-array-type host variable, the user must execute FETCH statement 1000 times to bring 1000 records. However, it is not necessary to communicate with the database server for each execution of FETCH statement. A certain number of records are brought from the database, and each record is stored in the host variable upon execution of FETCH statement. Therefore, using an array-type host variable during execution of FETCH statement will not enhance the performance. However, by lowering the execution count of FETCH statement, the user can expect performance improvement.

CONNTYPE and Host Arrary Variables

CONNTYPE

CONNTYPE is an option that decides the communication method with the database server. It can be specified upon establishment of a connection with the database server. CONNTYPE affects the performance and is closely related to array-type host variable.

CONNTYPE Type

Supported CONNTYPE includes TCP, UNIX and IPC. And it also includes DA method for embedded application.

The Relation between CONNTYPE and the Array Size of Host Variable

In general, the performance of CONNTYPE is high in order of IPC, UNIX, and TCP. However, when the input host variable is used as an array type, this is not always the case. Therefore, when the input host variable is used as the array type, the user can adjust CONNTYPE and array size suitable for the best performance.

Using Host Array Variables in SQL Statements

In various embedded SQL statements, array-type host variables can be used.

INSERT

The following shows the array types that can be used in INSERT statement:

Numeric-type or character-type arrays

Array type of the structure

Array type of the structure element

Example

The following is an example of using the array-type host variable as the input host variable in INSERT statement.

```
< Example Program : arrays1.sc >
EXEC SQL BEGIN DECLARE SECTION;
char a_gno[3][10+1];
char a_gname[3][20+1];
char a_goods_location[3][9+1];
int a_stock[3];
double a_price[3];
EXEC SQL END DECLARE SECTION;
strcpy(a_gno[0], "X111100001");
strcpy(a_gno[1], "X111100002");
strcpy(a_gno[2], "X111100003");
strcpy(a_gname[0], "XX-201");
strcpy(a_gname[1], "XX-202");
strcpy(a_gname[2], "XX-203");
strcpy(a_goods_location[0], "AD0010");
strcpy(a_goods_location[1], "AD0011");
strcpy(a_goods_location[2], "AD0012");
a_stock[0] = 1000;
a_stock[1] = 1000;
a_stock[2] = 1000;
a_price[0] = 5500.21;
a_price[1] = 5500.45;
a_price[2] = 5500.99;
EXEC SQL INSERT INTO GOODS
VALUES (:a_gno, :a_gname, :a_goods_location,
:a_stock, :a_price);
```

UPDATE

The following is the array type that can be used in UPDATE statement.

Numeric-type or character-type arrays

Array type of the structure element

Restrictions

The array type of the structure must not be used. The reason is that it is not possible to define array elements in the embedded SQL statement.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; int i3; } var1[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL UPDATE T1
SET I1 = :var1[0].i1, I2 = :var1[0].i2
WHERE I1 = :var1[0].i3;(X)
```

Example

The following is an example of using the array-type host variable as the input host variable in UPDATE statement.

```
< Example Program : arrays1.sc >
EXEC SQL BEGIN DECLARE SECTION;
int a_eno[3];
short a_dno[3];
char a_emp_tel[3][15+1];
EXEC SQL END DECLARE SECTION;
a_eno[0] = 10;
a_eno[1] = 11;
a_eno[2] = 12;
a_dno[0] = 2001;
a_dno[1] = 2001;
a_dno[2] = 2001;
strcpy(a_emp_tel[0], "01454112366");
strcpy(a_emp_tel[1], "0141237768");
strcpy(a_emp_tel[2], "0138974563");

EXEC SQL UPDATE EMPLOYEE
SET DNO = :a_dno,
 EMP_TEL = :a_emp_tel
 WHERE ENO = :a_eno;
```

DELETE

The following is an array type that can be used in DELETE statement.

Numeric-type or character-type arrays

Array type of the structure element

Restrictions

Like in UPDATE statement, the user cannot use an array type of the structure.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; int i3; } var1[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL DELETE FROM T1
WHERE I1 = :var1[0].i1 AND
I2 = :var1[0].i2 AND
I3 = :var1[0].i3;(X)
```

Example

The following is an example of using the array-type host variable as the input host variable in DELETE statement.

```
< Example Program : arrays1.sc >
EXEC SQL BEGIN DECLARE SECTION;
short a_dno[3];
EXEC SQL END DECLARE SECTION;
a_dno[0] = 4001;
a_dno[1] = 4002;
a_dno[2] = 2001;
EXEC SQL DELETE FROM EMPLOYEE
 WHERE DNO = :a_dno;
```

SELECT

The following shows the array types that can be used in SELECT statement. In FETCH statement, the following array types can be used and the same limitations will be applied.

Numeric-type or character-type arrays

Array type of the structure

Array type of the structure element

Restrictions

The input host variable must not be an array.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1;
int var2[10];
int var3[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT * INTO :var1
FROM T1 WHERE i1 = :var3;(X)
or
EXEC SQL SELECT * INTO :var2
FROM T1 WHERE i1 = :var3;(X)
```

In case the number of returned records is higher than the array size, "Returns too many rows" error will be displayed.

Example

The following is an example of using the array-type host variable as the output host variable in SELECT statement. At this time, the input host variable is not an array.

```
< Example Program : arrays2.sc >
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
short a_dno[5];
char a_dname[5][30+1];
char a_dep_location[5][9+1];
EXEC SQL END DECLARE SECTION;
s_dno = 3000;
```

```
EXEC SQL SELECT DNO, DNAME, DEP_LOCATION
INTO :a_dno, :a_dname, :a_dep_location
 FROM DEPARTMENT
WHERE DNO > :s_dno;
```

FOR Clause

Sometimes, only part of array components must be processes in the embedded SQL statement that uses an array-type input host variable. For example, when the data is fetched in Loop statement and the fetched data is inserted again, the final fetching count may be smaller than the array size. At this time, the processing count can be defined using FOR clause.

FOR clause is to set the number of arrays to process when an array-type input host variable is used.

FOR clause is prior to the array size of the host variable. For example, when the array size of the host variable is 10 and the count is specified as 5 in FOR clause, only five host variables from no. 0 to no. 4 will be processed.

In case the processing count changes for every execution of the embedded SQL statement, FOR clause is convenient to use.

The following is an embedded SQL statement that can use FOR clause.

- INSERT statement
- UPDATE statement
- DELETE statement

Syntax

```
EXEC SQL FOR <:host_var | constant> { INSERT … | UPDATE … | DELETE … }
```

Argument

<:*host_var*>: The processing count will be stored in host_var.

This host_var may not be declared in the DECLARE section of the host variable.

<*constant*>: The constant indicates the processing count.

Precautions

One or a higher value must be specified in FOR clause.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int cnt;
int var1[10];
int var2[10];
EXEC SQL END DECLARE SECTION;
cnt = 5;(O)
EXEC SQL FOR :cnt INSERT INTO T1
VALUES (:var1, :var2);
cnt = 0;(X)
EXEC SQL FOR :cnt INSERT INTO T1
```

```
VALUES (:var1, :var2);
cnt = -1;(X)
EXEC SQL FOR :cnt INSERT INTO T1
VALUES (:var1, :var2);
```

In case the input host variable is not an array type, FOR clause cannot be used.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int cnt;
int var1;
EXEC SQL END DECLARE SECTION;
cnt = 1;
EXEC SQL FOR :cnt INSERT INTO T1
VALUES (:var1); (X)
```

Example

Shows example of using FOR Clause in various SQL statements.

[Example 1] The following is an example of using FOR clause in INSERT statement. Specifies the processing count using host variable cnt, and inserts no. 0 and no. 1 values of a_goods in GOODS table.

< Example Program : hostvar.h >

```
XEC SQL BEGIN DECLARE SECTION;
typedef struct goods
{
 char gno[10+1];
 char gname[20+1];
 char goods_location[9+1];
 int stock;
 double price;
} goods;
EXEC SQL END DECLARE SECTION;
```

< Example Program : arrays1.sc >

```
EXEC SQL BEGIN DECLARE SECTION;
goods a_goods[3];
EXEC SQL END DECLARE SECTION;
int cnt;
cnt = 2;
EXEC SQL FOR :cnt INSERT INTO GOODS VALUES (:a_goods);
```

[Example 2] The following is an example of using FOR clause in UPDATE statement. The processing count must be a constant. Two cases from array no. 0 will be processed. In other words, DNO and EMP_TEL columns in records no. 0 and 1 in a_employee.eno will be changed into value no. 0 and 1 of a_employee.dno and a_employee.emp_tel respectively.

```
< Example Program : arrays1.sc >
EXEC SQL BEGIN DECLARE SECTION;
struct
{
 int eno[3];
 short dno[3];
 char emp_tel[3][15+1];
} a_employee;
EXEC SQL END DECLARE SECTION;
EXEC SQL FOR 2 UPDATE EMPLOYEE
SET DNO = :a_employee.dno,
 EMP_TEL = :a_employee.emp_tel,
JOIN_DATE = SYSDATE
WHERE ENO = :a_employee.eno;
```

[Example 3] The following is an example of using FOR clause in DELETE statement. The processing count must be specified by host variable cnt. Two cases from array no. 0 will be processed. In other words, only records no. 0 and 1 in a_dno will be deleted.

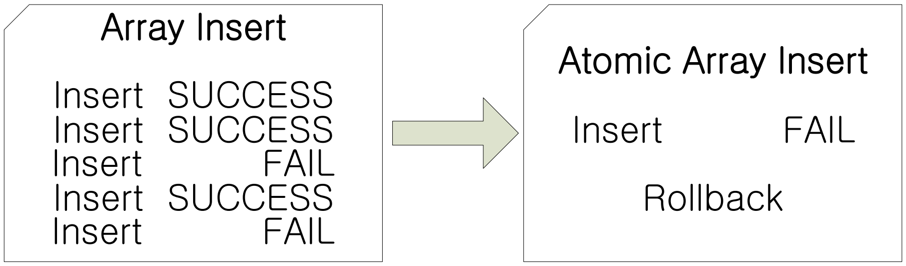
```
< Example Program : arrays1.sc >
EXEC SQL BEGIN DECLARE SECTION;
short a_dno[3];
EXEC SQL END DECLARE SECTION;
int cnt;
cnt = 2;
EXEC SQL FOR :cnt DELETE FROM EMPLOYEE
 WHERE DNO = :a_dno;
```

ATOMIC FOR Clause

If you use this in embedded SQL statement, where input host variables of array type are used, you can execute Atomic Array Insert which seems to process one stmt by binding several stmts.

Therefore, even if one stmt is failed when using this, all stmts are failed. You can't have several result values for each stmt but only one result value for several stmts.

Figure 10-1 Result Values of Array Insert and Atomic Array Insert



The existing Array Insert offers advantages to cut communication cost, but Atomic Array Insert has faster performance because it reduces the number of stmts additionally.

Table 10-1 Difference between Array Insert and Atomic Array Insert

Identification	Array Insert	Atomic Array Insert
The Number of Stmts	The Number of Arrays	One
The Number of Results	The Number of Arrays	One
Speed	Fast	Faster

Syntax

```
EXEC SQL ATOMIC FOR <:host_var | constant> {INSERT … }
```

Argument

<:*host_var*>: Its value indicates the number of its data processed. It doesn't have to be declared in the host variable declaration.

<*constant*>: This value indicates the number of this processed.

Restrictions

ATOMIC FOR clause is not used in other DML but only in INSERT statement. You can execute Atomic Array Insert for LOB column, but LOB data are not rollbacked if errors occur when they are transferred. Therefore, useres must get them rollbacked directly with savepoint. Additionally several restrictions to use Atomic For clause are as the following table.

Table 10-2 Restrictions on Atomic Array Insert

Identification	Array Insert	Atomic Array Insert
Foreign Key	Operating Normally	Operating Normally
Unique Key	Operating Normally	Operating Normally
Not Null	Operating Normally	Operating Normally
Trigger Each Row	Executed N times	Executed N times
Trigger Each Statement	Executed N times	Executed Once
Partitioned Table	Operating Normally	Operating Normally
Sequence	Executed N times	Executed N times
SYSDATE	Executed N times	Executed Once
LOB Column	Operating Normally	Atomic attribute is not guaranteed when errors occur.
Procedure	Operating Normally	Operating Normally
SubQuery	Always Looking at New View	Looking at View Executed First

Example

```
EXEC SQL ATOMIC FOR 20 INSERT INTO T1 VALUES( :data );
```

ONERR Clause

If checking the embedded SQL statements which use array typed host variable, you can know whether array elements make up an array successfully. Therefore, if some array elements fail to do, you can manage them by writing DML.

Syntax

```
EXEC SQL ONERR <:host_var , :host_var> {INSERT | UPDATE | DELETE}
```

or

```
EXEC SQL ONERR <:host_var , :host_var> FOR <:host_var | constant>
{INSERT | UPDATE | DELETE}
```

Argument

<:host_var , :host_var>: This denotes to save result value of executing SQL statements for the first host variable and result value is specified as short type. Error code is saved for the second host variable and is specified as int type.

<:*host_var*>: This value denotes the number of its data processed. This doesn't have to be declared in the host variable declaration.

<*constant*>: This value denotes the number of its data processed.

Restriction

- Array size of host variable used in ONERR clause should not be lesser than that used in SQL statements.
- If you use array size of host variable, which is used in ONERR clause, in FOR clause, array size should not be lesser than that specified in DML statement.

Example

```
EXEC SQL ONERR :err_rc , :err_code UPDATE T1 SET c2 = c2+1 WHERE c1 = :var1;
EXEC SQL ONERR :err_rc , :err_code FOR :arr_count UPDATE T1 SET c2 = c2+1
WHERE c1 = :var1;
```

sqlca.sqlerrd

In case an array-type host variable is used in the embedded SQL statement, the user can refer to sqlca.sqlerrd variable after executing embedded SQL statement. The following describes which kind of execution result is stored in the sqlca.sqlerrd variable.

sqlca.sqlerrd[2]

If the host variable is not an array type, the user can refer to this value after executing UPDATE statement and DELETE statement.

If the host variable is an array type, the user can refer to this value after executing INSERT statement, UPDATE statement, DELETE statement, and SELECT statement.

The following describes the meaning of sqlca.sqlerrd[2] variable when sqlca.sqlcode is SQL_SUCCESS after executing each embedded SQL statement.

INSERT

If the host variable is not an array type, this value will be 1. It means, one record has been inserted.

In case of the array type, the success count will be stored. Therefore, this value must not be larger than the array size.

For example, if the user executed INSERT using a host variable of which array size is 3 and succeeded three times, this value will be 3. If the user succeeds two times and fails once in the last time, this value will be two.

However, if the host variable is an array type, this value indicates the number of arrays specified when you succeed to execute Atomic Array Insert completely. Otherwise, this value will be 0.

UPDATE/DELETE

The number of updated or deleted records will be stored.

There can be more than one record meeting the conditions of the host variable, this value may be higher than the array size.

For example, assume that you have updated using a host variable of which array size is 3 and succeeded three times. If there were three records meeting the condition no. 0, two records meeting the condition no. 1, and zero record meeting the condition no. 2, total five records will be updated and this value will be 5.

SELECT/FETCH

If the output host variable is not an array type, this value will be meaningless (garbage value.)

In case of an array type, the number of currently selected (or fetched) records will be stored. In other words, when fetching is made several times, the number of only currently fetched records will be stored. The records will not be accumulated. Therefore, this value must not be larger than the array size.

If the number of the returned records is same as or smaller than the array size, the execution result (sqlca.sqlcode) will be SQL_SUCCESS, and the number of returned records will be stored in sqlca.sqlerrd[2].

If there is no returned record, the execution result (sqlcal.sqlcode) will be SQL_NO_DATA and "0" will be stored in sqlca.sqlerrd[2].

For example, assume that you have executed SELECT using a host variable of which array size is 10 as an output host variable. If there are five records meeting the conditions, give records from array no. 0 will be stored in the output host variable in order. At this time the execution result (sqlca.sqlcode) will be SQL_SUCCESS, and "5" will be stored in sqlca.sqlerrd[2].

sqlca.sqlerrd[3]

In this variable, the success count after executing of the embedded SQL statement using array-type input host variable is stored. Therefore, this value must not be larger than the array size.

To refer to this variable, the following conditions must be met:

Sqlca.sqlcode must be SQL_SUCCESS.

Execute the embedded SQL statement using an array-type input host variable before referring.

Execute INSERT, UPDATE, DELETE or SQL/PSM statement before referring.

If you are successful in executing Atomic Array Insert, its value is 1. If you are failed to do, its value is 0.

Example

Assume that you have updated the data using a host variable of which array size is 3. If updating was successful with array no. 0 and no. 1 value while updating was failure using array no. 2 value, "2" will be store din sqlca.sqlerrd[3]. If there are three records meeting the condition no. 0 and two records meeting the condition no. 1, "5" will be stored in sqlca.sqlerrd[2.

Precautions

Unless sqlca.sqlcode is SQL_SUCESS, sqlca.sqlerrd variable will not have any meaning (garbage value.) Therefore, when sqlca.sqlcode is SQL_SUCCESS, refer to sqlca.sqlerrd variable.

When using an array-type host variable in AUTOCOMMIT mode, each array element, not an entire array, is one transaction. Therefore, if only some elements are successful while others are not, the changes in the successful transaction are permanently stored in the database server.

For example, if inserting is successful with array values no. 0 and no. 1 while inserting was a failure with array value no. 2 when the host variable of which array size is 3 is used, the two successful cases will be inserted in the table.

Considerations for Using Host Array Variables

There are several considerations in relation to the use of the array-type host variable: Note the following when writing a program:

In the DECLARE section

The pointer type cannot be declared by the array.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
char *var1[10];(X)
EXEC SQL END DECLARE SECTION;
```

Only a single-dimensional array is allowed. In some cases, two-dimensional arrays are allowed for 'char' type and 'varchar' type.

```
Correct Example) EXEC SQL BEGIN DECLARE SECTION;
char var1[10][10];
int var2[10];
EXEC SQL END DECLARE SECTION;



Incorrect Example) EXEC SQL BEGIN DECLARE SECTION;
char var3[10][10][10];
int var4[10]10];
EXEC SQL END DECLARE SECTION;
```

In SQL Statements

No array element can be set.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1[10];
int var2[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1
VALUES (:var1[0], :var2[0]);(X)
```

The input host variables of SELECT statement and CURSOR statement must not be an array.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1[10];
int var2[10];
int var3[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT I1, I2 INTO :var1, :var2
FROM T1 WHERE I1 = :var3;(X)
```

Host Structures and Arrays

The user can declare the array or structure components as arrays using the host variable.

Array of the Structure

Declares the structure type as an array.

Advantages

It is convenient to use INSERT statement when inserting multiple records into all columns of one table.

It is convenient to use SELECT or FETCH statement to bring multiple columns at the same time from one table.

Weakness

As the indicator variable cannot be specified, this cannot be used when the input value is NULL or the selected or fetched value is NULL.

As the array component cannot be specified, this cannot be used in SET clause of UPDATE statement or WHERE clause of DML statement.

Restrictions

Only a single-dimensional array is allowed. In case of the array of the structure, the component of the corresponding structure cannot become an array.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1[10]; int i2[10]; } var1[10];(X)
EXEC SQL END DECLARE SECTION;
As the array component cannot be referred to from the embedded SQL statement,
the components of the structure cannot be referred to either.
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2)
VALUES(:var1[0].i1, :var1[0].i2);(X)
EXEC SQL INSERT INTO T1(I1, I2)
VALUES(:var1); (O)
```

In case the structure array is used as a host variable in INTO clause of SELECT or FETCH statement, only one output host variable can be used. In other words, it must not be used with other host variables. Therefore, if the host variable to be used in INTO clause is a structure, the number of components must be the same as the number of the columns in SELECT clause.

In case the structure array is used as a host variable in VALUES clause of INSERT statement, only one input host variable must be used. In other words, it must not be used with other host variables. Therefore, if the host variable to be used in VALUES clause is an array type of the structure, the number of the elements of this structure must be the same as the number of columns in INSERT statement.

This is due to an internal rule that requires the structure to include all host variables when the host variable is an array of the structure.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1[10];
struct tag1 { int i3; int i4; } var2[10];
int var3;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT I1, I2 INTO :var1
FROM T1 WHERE I1 = :var3;(O)
EXEC SQL SELECT I1, I2, I3, I4 INTO :var1, :var2
FROM T1 WHERE I1 = :var3;(X)
```

The indicator variable cannot be specified. Therefore, when this type is used as an output host variable, it must be guaranteed that the returned column is not NULL.

This is due to an internal rule that requires the structure to include all host variables and indicator variables when the host variable is an array of the structure.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1[10];
struct tag2 { int i1_ind; int i2_ind; } var1_ind[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT * INTO :var1 :var1_ind;(X)
```

As the component cannot be specified for the input host variable, this can be used only in INSERT statement.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; } var1[10];
int var3;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO T1(I1, I2)
VALUES(:var1);(O)
EXEC SQL UPDATE T1
SET I1 = :var1[0].i1,
I2 = :var1[0].i2
```

Example

Shows various examples of declaring the array type of the structure as the host variable and using it.

[Example 1] The following is an example of using the array-type host variable of the structure as the input host variable in INSERT statement.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct goods
{
 char gno[10+1];
 char gname[20+1];
 char goods_location[9+1];
 int stock;
 double price;
} goods;
EXEC SQL END DECLARE SECTION;
```

< Example Program : arrays1.sc >

```
/* specify path of header file */
```

```
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
goods a_goods[3];
EXEC SQL END DECLARE SECTION;
strcpy(a_goods[0].gno, "Z111100001");
strcpy(a_goods[1].gno, "Z111100002");
strcpy(a_goods[2].gno, "Z111100003");
strcpy(a_goods[0].gname, "ZZ-201");
strcpy(a_goods[1].gname, "ZZ-202");
strcpy(a_goods[2].gname, "ZZ-203");
strcpy(a_goods[0].goods_location, "AD0020");
strcpy(a_goods[1].goods_location, "AD0021");
strcpy(a_goods[2].goods_location, "AD0022");
a_goods[0].stock = 3000;
a_goods[1].stock = 4000;
a_goods[2].stock = 5000;
a_goods[0].price = 7890.21;
a_goods[1].price = 5670.45;
a_goods[2].price = 500.99;
EXEC SQL INSERT INTO GOODS VALUES (:a_goods);
```

[Example 2] The following is an example of using the array-type host variable of the structure as the output host variable in SELECT statement.

< Example Program : hostvar.h >

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct department
{
 short dno;
 char dname[30+1];
 char dep_location[9+1];
 int mgr_no;
} department;
EXEC SQL END DECLARE SECTION;
```

< Example Program : arrays2.sc >

```
/* specify path of header file */
EXEC SQL OPTION (INCLUDE=./include);
/* include header file for precompile */
EXEC SQL INCLUDE hostvar.h;
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
department a_department[5];
EXEC SQL END DECLARE SECTION;
s_dno = 2000;
EXEC SQL SELECT * INTO :a_department
 FROM DEPARTMENT WHERE DNO < :s_dno;
```

Array of Structs

Declares the component of the structure as arrays.

Advantages

It is convenient to use INSERT statement when inserting multiple records into all columns of one table.

It is convenient to use SELECT or FETCH statement to bring multiple columns at the same time from one table.

The indicator variable can be specified, and NULL data can be processed.

As the component of the structure can be specified, it can be used in UPDATE statement or WHERE clause of DML statement.

While the array type of the structure can use only the host variable in the input host variable list or output host variable list, the structure type of which component is an array can use multiple host variables in the input host variable list or output host variable list.

Example

Shows various examples of declaring components of the structure as arrays and using them.

[Example 1] The following is an example of using the host variable of the structure type of which component is an array as an input host variable in UPDATE statement. Specifies SQL_NULL_DATA in a_emp_tel_ind, and changes EMP_TEL column into NULL data.

```
< Example Program : arrays1.sc >
EXEC SQL BEGIN DECLARE SECTION;
struct
{
 int eno[3];
 short dno[3];
 char emp_tel[3][15+1];
} a_employee;
int a_emp_tel_ind[3];
EXEC SQL END DECLARE SECTION;
/* set host variables */
a_employee.eno[0] = 17;
a_employee.eno[1] = 16;
a_employee.eno[2] = 15;
a_employee.dno[0] = 1003;
a_employee.dno[1] = 1003;
a_employee.dno[2] = 1003;
/* set indicator variables */
a_emp_tel_ind[0] = SQL_NULL_DATA;
a_emp_tel_ind[1] = SQL_NULL_DATA;
a_emp_tel_ind[2] = SQL_NULL_DATA;
EXEC SQL UPDATE EMPLOYEE
SET DNO = :a_employee.dno,
 EMP_TEL = :a_employee.emp_tel :a_emp_tel_ind,
 JOIN_DATE = SYSDATE
 WHERE ENO > :a_employee.eno;
```

[Example 2] The following is an example of using the host variable of the structure type of which component is an array as an output host variable in SELECT statement.

```
< Example Program : arrays2.sc >
EXEC SQL BEGIN DECLARE SECTION;
short s_dno;
struct
{
 short dno[5];
 char dname[5][30+1];
 char dep_location[5][9+1];
 int mgr_no[5];
} a_department2;
EXEC SQL END DECLARE SECTION;
s_dno = 2000;
EXEC SQL SELECT * INTO :a_department2
FROM DEPARTMENT WHERE DNO < :s_dno;
```

Sample Programs

arrays1.sc

```
See $ALTIBASE_HOME/sample/APRE/arrays1.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make arrays1
shell> ./arrays1
<ARRAYS 1>
----------------------------------------------------
[Scalar Array Host Variables With Insert]
----------------------------------------------------
3 rows inserted
3 times insert success
----------------------------------------------------
[Structure Array Host Variables With Insert]
----------------------------------------------------
3 rows inserted
3 times insert success
----------------------------------------------------
[Arrays In Structure With Insert]
----------------------------------------------------
3 rows inserted
3 times insert success
----------------------------------------------------
[Error Case : Array Host Variables With Insert]
----------------------------------------------------
SQLCODE : -69720
sqlca.sqlerrm.sqlerrmc : [ROW-2] ERR-11058(23000) : The row already exists in
a unique index.
[ROW-3] ERR-11058(23000) : The row already exists in a unique index.
1 rows inserted
1 times insert success
----------------------------------------------------
[Scalar Array Host Variables With Update]
----------------------------------------------------
3 rows updated
3 times update success
----------------------------------------------------
[Arrays In Structure With Update]
----------------------------------------------------
12 rows updated
3 times update success
----------------------------------------------------
[Scalar Array Host Variables With Delete]
----------------------------------------------------
6 rows deleted
3 times delete success
----------------------------------------------------
[For Clause With Insert]
----------------------------------------------------
2 rows inserted
2 times insert success
----------------------------------------------------
[For Clause With Update]
----------------------------------------------------
```

```
2 rows updated
2 times update success
----------------------------------------------------
[For Clause With Delete]
----------------------------------------------------
3 rows deleted
2 times delete success
```

arrays2.sc

```
See $ALTIBASE_HOME/sample/APRE/arrays2.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make arrays2
shell> ./arrays2
<ARRAYS 2>
----------------------------------------------------
[Scalar Array Host Variables With Select]
----------------------------------------------------
DNO DNAME DEP_LOCATION
----------------------------------------------------
3001 CUSTOMER SUPPORT DEPT London
3002 PRESALES DEPT Peking
4001 MARKETING DEPT Seoul
4002 BUSINESS DEPT LA
4 rows selected
----------------------------------------------------
[Structure Array Host Variables With Select]
----------------------------------------------------
DNO DNAME DEP_LOCATION MGR_NO
----------------------------------------------------
1001 RESEARCH DEVELOPMENT DEPT 1 New York 16
1002 RESEARCH DEVELOPMENT DEPT 2 Sydney 13
1003 SOLUTION DEVELOPMENT DEPT Japan 14
3 rows selected
----------------------------------------------------
[Arrays In Structure With Select]
----------------------------------------------------
DNO DNAME DEP_LOCATION MGR_NO
----------------------------------------------------
1001 RESEARCH DEVELOPMENT DEPT 1 New York 16
1002 RESEARCH DEVELOPMENT DEPT 2 Sydney 13
1003 SOLUTION DEVELOPMENT DEPT Japan 14
3 rows selected
----------------------------------------------------
[Error Case : Array Host Variables]
----------------------------------------------------
Error : [-594092] Returns too many rows
----------------------------------------------------
[Execute Procedure With Array In-Binding]
----------------------------------------------------
Success execute procedure
```

Part III

11 Dynamic SQL Statements

Static Versus Dynamic SQL Statements

Static SQL statements

Concepts

The SQL statement to be used is defined and fixed by the programmer in advance.

The decided SQL statement is hard-coded in the program like a series of embedded SQL statements.

For more information about the static SQL statement, see Chapters V and VII.

Limitations

Cannot be used in a program in which the SQL statement is not pre-determined.

The tables and columns to be referred must be decided by the programmer in advance. In other words, the host variable cannot replace the table or column name.

The input host variable of the static SQL statement provides a little bit of flexibility but it is fixed in nature.

Dynamic SQL statements

Concepts

Configures the text of the SQL statement in a data area in the program running time. Therefore, the SQL statement does not need to be hard-coded in the program source code.

Advantages

The programmer does not need to decide the SQL statement in program according to application logic. In other words, the dynamic SQL statement is available.

The tables and columns to be referred to can be dynamically decided upon execution.

Weakness

As the SQL statement to be used as well as the table and columns to be referred to only in the execution time, it may be less effective than the static SQL statement in terms of performance.

Using Dynamic SQL Statements

Method 1

Syntax

```
EXEC SQL EXECUTE IMMEDIATE

<:host_var | string_literal>;
```

Argument

<:host_var> : Variable including the entire strings of the SQL statement

<string_literal> : Entire string of the SQL statement

Description

When using a host variable, input only one host variable. This host variable must include the all strings of the SQL statement. The strings of the SQL statement must not include parameter marketers that will be replaced with the host variables.

In case this method is used for a frequently used SQL statement, it may compromise the performance.

Effective for the DDL statement in which the SQL statement is decided on the execution time.

SELECT statement cannot be used.

Example

[Example 1] The following is an example of dynamic SQL statement method1 using SQL statement string.

```
< Example Program : dynamic1.sc >
EXEC SQL EXECUTE IMMEDIATE
DROP TABLE T1;
EXEC SQL EXECUTE IMMEDIATE
CREATE TABLE T1 (I1 INTEGER, I2 INTEGER);
```

[Example 2] The following is an example of dynamic SQL statement method1 using the host variable.

```
< Example Program : dynamic1.sc >
char query[100];
strcpy(query, "drop table t2");
EXEC SQL EXECUTE IMMEDIATE :query;
strcpy(query, "create table t2(i1 integer)");
EXEC SQL EXECUTE IMMEDIATE :query;
```

Method 2

method2 is consist of two phases of PREPARE statement and EXECUTE statement. This method is to execute the prepared SQL statement several times.

The SQL statement string to be used in PREPARE statement must not include ? (parameter marker) which can be replaced with the host variable of EXECUTE statement.

If the prepared SQL statement is changed between PREPARE statement and EXECUTE statement, EXECUTE statement will use the SQL statement used in the previous PREPARE statement until PREPARE statement is executed again.

Frequent changing of SQL statement will compromise execution performance of PREPARE and EXECUTE statements.

Effective for INSERT, UPDATE, and DELETE statement for which the SQL statement is decided on execution time.

SELECT statement cannot be used.

PREPARE

Syntax

```
EXEC SQL PREPARE <statement_name> FROM

<:host_var | string_literal>;
```

Argument

<statement_name> : SQL statement identifier. Up to maximum 50 characters

<:host_var> : Variable including the entire strings of the SQL statement

<string_literal> : Entire string of the SQL statement

Description

Prepare for execution of the SQL statement.

In case the same SQL statement identifier is used in multiple PREPARE statements of one program, the SQL of the latest PREPARE statement will be executed when EXECUTE statement is called on the execution time.

SQL statement cannot function as SELECT statement.

Example

The following is an example showing that the SQL statement is decided at the execution time according to the conditions and the preparation is made for the corresponding SQL statement.

```
< Example Program : dynamic2.sc >
char query[100];
EXEC SQL BEGIN DECLARE SECTION;
```

```
int s_eno;
EXEC SQL END DECLARE SECTION;
if (s_eno < 20)
{
 strcpy(query, "delete from employee where eno = ? and ename = ?");
}
else
{
 strcpy(query, "insert into employee(eno, ename) values (?, ?)");
}
EXEC SQL PREPARE S FROM :query;
```

EXECUTE

Syntax

```
EXEC SQL EXECUTE <statement_name>

[ USING <host_var_list> ];
```

Argument

<statement_name> : SQL statement identifier

<host_var_list> : List of input host variables and input indicator variables

Description

Can be executed after PREPARE statement. When EXECUTE statement is executed with an undefined SQL statement identifier, "Not defined" error message will be displayed.

The SQL statement of the defined SQL statement identifier cannot function as SELECT statement.

Executes a prepared SQL statement.

Delivers the value to each parameter marker. Set the host variable list in USING clause to send the values to the parameter marker. At this time, the number of the host variables in USING clause must be the same as the number of the parameter markers in the SQL statement. The host variable type must be convertible with the corresponding column type.

In case the same SQL statement identifier is used for multiple PREPARE statements in one program, the SQL statement of the latest PREPARE statement will be executed on the execution time.

Example

The following is an example of EXECUTE statement.

```
< Example Program : dynamic2.sc >
EXEC SQL BEGIN DECLARE SECTION;
int s_eno;
char s_ename[20+1];
EXEC SQL END DECLARE SECTION;
s_eno = 10;
strcpy(s_ename, "YHBAE");
EXEC SQL EXECUTE S USING :s_eno, :s_ename;
```

Method 3

method3 consists of five phases of PREPARE statement, DECLARE statement, OPEN statement, FETCH statement, and CLOSE statement.

Only SELECT statement can be used.

PREPARE

Syntax

```
EXEC SQL PREPARE <statement_name> FROM

<:host_var | string_literal>;
```

Argument

<statement_name> : SQL statement identifier. Up to maximum 50 characters

<:host_var> : Variable including the entire strings of the SQL statement

<string_literal> : Entire string of the SQL statement

Description

Prepare for execution of the SQL statement.

In case the same SQL statement identifies user used in multiple PREPARE statements in one program, the SQL statement of the latest PREPARE statement will be used to declare the cursor when DECLARE statement is called on the execution time.

SQL statement must SELECT statement.

Example

The following is an example showing that the SQL statement is decided at the execution time according to the conditions and the preparation is made for the corresponding SQL statement.

```
< Example Program : dynamic3.sc >
char query[100];
int type;
switch (type)
{
case 1:
 strcpy(query, "select * from department");
 break;
case 2:
 strcpy(query, "select * from goods");
 break;
case 3:
 strcpy(query, "select * from orders");
 break;
}
EXEC SQL PREPARE S FROM :query;
```

DECLARE CURSOR

Syntax

```
EXEC SQL DECLARE <cursor_name> CURSOR FOR

<statement_name>;
```

Argument

<cursor_name> : Cursor name

<statement_name> : SQL statement identifier

Description

Can be executed after PREPARE statement, CLOSE statement, and CLOSE RELEASE statement. In case Cursor Declaration statement is executed with an undefined SQL statement identifier, "Not defined" error message will be displayed.

The SQL statement of the defined SQL statement identifier must be SELECT statement.

Declare the cursor using the defined SQL statement identifier.

This SQL statement is ready for execution.

In case the same SQL statement identifies user used in multiple PREPARE statements in one program, the SQL statement of the latest PREPARE statement will be used to declare the cursor.

Example

In the following example, cursor CUR is declared by SQL statement identifier S.

```
< Example Program : dynamic3.sc >
EXEC SQL DECLARE CUR CURSOR FOR S;
```

OPEN

Syntax

```
EXEC SQL OPEN <cursor_name>

[ USING <host_var_list> ];
```

Argument

<cursor_name> : Cursor name

<host_var_list> : Output host variable and output indicator variable list

Description

Cursor OPEN statement can be executed after Cursor Declaration statement or Cursor CLOSE statement. In case an undefined cursor is opened, "Not defined" error message will be displayed.

Cursor OPEN statement executes SQL statement of Cursor Declaration statement.

Delivers the value to each parameter marker. Set the host variable list in USING clause to send the values to the parameter marker. At this time, the number of the host variables in USING clause must be the same as the number of the parameter markers in the SQL statement. The host variable type must be convertible with the corresponding column type.

In case the same cursor name is declared more than once, refer to the cursor with the most recent execution time.

Restrictions

The restrictions in SELECT statement is applied.

Example

The following is an example of opening cursor CUR.

```
< Example Program : dynamic3.sc >
```

EXEC SQL OPEN CUR;

FETCH

Syntax

```
EXEC SQL FETCH <cursor_name> INTO <host_var_list>;
```

Argument

<cursor_name> : Cursor name

<host_var_list> : Output host variable and output indicator variable list

Description

Cursor FETCH statement can be executed after Cursor OPEN statement. In case an unopened cursor is fetched, "Function sequence error" error message will be displayed.

Used to receive the result of opening the cursor. To receive the result, define the host variable list in INTO clause. At this time, the number of host variables must same with the number of columns in SELECT clause. Each host variable type must be compatible with the corresponding column type.

In case the same cursor name is declared more than once, refer to the cursor with the most recent execution time.

Example

The following example shows how to fetch while changing the output host variable of cursor CUR depending on the conditions. by defining the indicator variable for each host variable, the user can check NULL data. Fetches till SQL_NO_DATA is returned in While loop.

```
< Example Program : dynamic3.sc >
int type;
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
/*declare output host variables */
department s_department;
goods s_goods;
orders s_orders;
/*declare indicator variables */
dept_ind s_dept_ind;
good_ind s_good_ind;
order_ind s_order_ind;
EXEC SQL END DECLARE SECTION;
while(1)
{
 /* use indicator variables to check null value */
 switch (type)
 {
 case 1:
 EXEC SQL FETCH CUR
INTO :s_department :s_dept_ind;
 break;
 case 2:
 EXEC SQL FETCH CUR
INTO :s_goods :s_good_ind;
 break;
 case 3:
 EXEC SQL FETCH CUR
INTO :s_orders :s_order_ind;
 break;
 }
 if (sqlca.sqlcode == SQL_SUCCESS)
{
 cnt++;
 }
 else if (sqlca.sqlcode == SQL_NO_DATA)
 {
 printf("%d rows selected\n\n", cnt);
 break;
 }
 else
 {
 printf("Error : [%d] %s\n\n",
SQLCODE, sqlca.sqlerrm.sqlerrmc);
 break;
 }
}
```

CLOSE

Syntax

```
EXEC SQL CLOSE <cursor_name>;
```

Argument

<cursor_name> : Cursor name

Description

Cursor CLOSE statement can be executed after Cursor Declaration statement, Cursor OPEN statement, and Cursor FETCH statement. In case an undefined cursor is closed, "Not defined" error message will be displayed.

In case fetching is not fully made in Cursor CLOSE statement, the result in the server will be deleted. At this time, the resources allocated to the cursor will not be cancelled. Therefore, after executing Cursor CLOSE statement, the user can immediately execute OPEN statement omitting Cursor Declaration statement.

In case the same cursor name is declared more than once, refer to the cursor with the most recent execution time.

Example

The following is an example of closing cursor CUR.

```
< Example Program : dynamic3.sc >
EXEC SQL CLOSE CUR;
```

CLOSE RELEASE

Syntax

```
EXEC SQL CLOSE RELEASE <cursor_name>;
```

Argument

<cursor_name> : Cursor name

Description

Cursor CLOSE RELEASE statement can be executed after Cursor Declaration statement, Cursor OPEN statement, Cursor FETCH statement, and Cursor CLOSE statement. In case an undefined cursor is close-released, "Not defined" error message will be displayed.

In case fetching is not fully made in Cursor CLOSE RELEASE statement, the result in the server will be deleted. Also, all resources allocated to the cursor will be cancelled. Therefore, after executing Cursor CLOSE RELEASE statement, the user must execute Cursor Declaration statement and Cursor OPEN statement in order. In other words, after Cursor CLOSE RELEASE statement, the user cannot execute Cursor OPEN statement.

To use the cursor again, execute CLOSE statement. Otherwise, execute CLOSE RELEASE statement. As the cursor is usually used again, Cursor CLOSE RELEASE statement is rarely executed. If the user selects CLOSE RELEASE -> Cursor Declaration statement -> Cursor OPEN statement for the same cursor to use again, the performance will be compromised.

In case the same cursor name is declared more than once, refer to the cursor with the most recent execution time.

Example

The following example is to CLOSE cursor CUR and cancel resources allocated to CUR.

```
EXEC SQL CLOSE RELEASE CUR;
```

Sample Programs

dynamic1.sc

```
$ALTIBASE_HOME/sample/APRE/dynamic1.sc
```

Execution Result

```
shell> is -f schema/schema.sql
shell> make dynamic1
shell> ./dynamic1
<DYNAMIC SQL METHOD 1>
----------------------------------------------------
[Using String Literal]
----------------------------------------------------
Success execution with string literal
----------------------------------------------------
[Using Host Variable]
----------------------------------------------------
Success execution with host variable
```

dynamic2.sc

See $ALTIBASE_HOME/sample/APRE/dynamic2.sc

Execution Result

```
shell> make dynamic2
shell> ./dynamic2
<DYNAMIC SQL METHOD 2>
----------------------------------------------------
[Prepare]
----------------------------------------------------
Success prepare
----------------------------------------------------
[Execute]
----------------------------------------------------
Success execute
```

dynamic3.sc

See $ALTIBASE_HOME/sample/APRE/dynamic3.sc

Execution Result

```
shell> is –f schema/schema.sql
shell> make dynamic3
shell> ./dynamic3
<DYNAMIC SQL METHOD 3>
----------------------------------------------------
[Prepare]
```

```
----------------------------------------------------
Success prepare
----------------------------------------------------
[Declare Cursor]
----------------------------------------------------
Success declare cursor
----------------------------------------------------
[Open Cursor]
----------------------------------------------------
Success open cursor
----------------------------------------------------
[Fetch Cursor]
----------------------------------------------------
30 rows selected
----------------------------------------------------
[Close Cursor]
----------------------------------------------------
Success close cursor
```

12 Using Stored Procedures in C/C++

Using Stored Procedures

Stored procedure or stored function-related embedded SQL statements are called stored procedure-processing SQL statements.

The user can create and execute the stored procedure or stored function in the application.

CREATE

Creates a stored procedure or stored function.

Syntax

stored procedure

```
EXEC SQL CREATE [ OR REPLACE ] PROCEDURE
<procedure_name>
[ ( [ <parameter_declaration_list> ] ) ]
AS | IS
[ <declaration_section> ]
BEGIN <statement>
[ EXCEPTION <exception_handler> ]
END
[ <procedure_name> ] ;
END-EXEC;
```

stored functions

```
EXEC SQL CREATE [ OR REPLACE ] FUNCTION
<function_name>
[ ( [ <parameter_declaration_list> ] ) ]
RETURN <data_type>
AS | IS
[ <declaration_section> ]
BEGIN <statement>
[ EXCEPTION <exception_handler> ]
END
[ <function_name> ] ;
END-EXEC;
```

Arguments

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

<parameter_declaration_list> : See Stored Procedure User’s Manual

<declaration_section> : See Stored Procedure User’s Manual

<statement> : See Stored Procedure User’s Manual

<exception_handler> : See Stored Procedure User’s Manual

<data_type> : See Stored Procedure User's Manual

Example

Displays an example of creating a stored procedure or stored function.

[Example 1] The following is an example of creating a stored procedure.

Searches the records in which ONO column is same as s_ono, the parameter of ORDER_PROC. If the search result is 0, inserts new records in ORDERS table using defined variables. If the search result is 1 or higher, changes the columns of the searched record into the defined variables.

```
< Example Program : psm1.sc >
EXEC SQL CREATE OR REPLACE PROCEDURE ORDER_PROC
(s_ono in bigint)
 AS
 p_order_date date;
 p_eno integer;
 p_cno bigint;
 p_gno char(10);
 p_qty integer;
 BEGIN
 SELECT ORDER_DATE, ENO, CNO, GNO, QTY
 INTO p_order_date, p_eno, p_cno, p_gno, p_qty
 FROM ORDERS
 WHERE ONO = s_ono;
 EXCEPTION
 WHEN NO_DATA_FOUND THEN
 p_order_date := SYSDATE;
 p_eno := 13;
 p_cno := BIGINT'7610011000001';
 p_gno := 'E111100013';
 p_qty := 4580;
 INSERT INTO ORDERS
(ONO, ORDER_DATE, ENO,
CNO, GNO, QTY)
VALUES
(s_ono, p_order_date, p_eno,
p_cno, p_gno, p_qty);
 WHEN OTHERS THEN
 UPDATE ORDERS
 SET ORDER_DATE = p_order_date,
 ENO = p_eno,
 CNO = p_cno,
 GNO = p_gno,
 QTY = p_qty
 WHERE ONO = s_ono;
 END;
 END-EXEC;
```

[Example 2] The following is an example of creating a stored function.

Inserts new records in ORDERS table, searches the total number of records in ORDERS table, and returns the result.

```
< Example Program : psm2.sc >
EXEC SQL CREATE OR REPLACE FUNCTION ORDER_FUNC(
 s_ono in bigint, s_order_date in date,
 s_eno in integer, s_cno in char(13),
 s_gno in char(10), s_qty in integer)
 RETURN INTEGER
 AS
 p_cnt integer;
 BEGIN
 INSERT INTO ORDERS
(ONO, ORDER_DATE,
ENO, CNO, GNO, QTY)
VALUES
(s_ono, s_order_date, s_eno,
 s_cno, s_gno, s_qty);
 SELECT COUNT(*)
INTO p_cnt
FROM ORDERS;
 RETURN p_cnt;
 END;
 END-EXEC;
```

ALTER

Precompiles the stored procedure or stored function.

Syntax

stored procedure

```
EXEC SQL ALTER PROCEDURE <procedure_name> COMPILE;
```

stored functions

```
EXEC SQL ALTER FUNCTION <function_name> COMPILE;
```

Argument

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

Description

Precompiles an invalid stored procedure or stored function, and makes it valid.

Example

Displays an example of precompiling the stored procedure or stored function.

[Example 1] The following is an example of precompiling ORDER_PROD stored procedure.

```
EXEC SQL ALTER PROCEDURE ORDER_PROC COMPILE;
```

[Example 2] The following show how to delete ORDER_FUNC stored function.

```
EXEC SQL ALTER FUNCTION ORDER_FUNC COMPILE;
```

DROP

Deletes the stored procedure or stored function.

Syntax

Stored procedure

```
EXEC SQL DROP PROCEDURE <procedure_name>;
```

Stored functions

```
EXEC SQL DROP FUNCTION <function_name>;
```

Argument

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

Example

Displays an example of deleting the stored procedure or stored function.

[Example 1] The following show how to delete ORDER_FUNC stored function.

```
< Example Program : psm1.sc >
EXEC SQL DROP PROCEDURE ORDER_PROC;
```

[Example 2] The following show how to delete ORDER_FUNC stored function.

```
< Example Program : psm2.sc >
EXEC SQL DROP FUNCTION ORDER_FUNC;
```

EXECUTE

Execute the stored procedure or stored function.

Syntax

Stored procedure

```
EXEC SQL EXECUTE BEGIN
<procedure_name>
[ ( [ <:host_var> [ IN | OUT | IN OUT ] [ ,
… ] ] ) ];
END;
```

```
END-EXEC;
```

Stored functions

```
EXEC SQL EXECUTE BEGIN
<:host_var> := <function_name>
[ ( [ <:host_var> [ IN | OUT | IN
OUT ] [ , … ] ] ) ];
END;
END-EXEC;
```

Argument

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

<:host_var> : Uses the parameter or input host variable necessary for the execution of the stored procedure.

Uses the parameter or input host variable necessary for the execution of the stored function.

Input host variable to store the result of the stored function.

Example

Display an example of executing the stored procedure or stored function.

[Example 1] The following is an example of executing stored procedure ORDER_PROC.

```
< Example Program : psm1.sc >
EXEC SQL BEGIN DECLARE SECTION;
long long s_ono;
EXEC SQL END DECLARE SECTION;
s_ono = 111111;
EXEC SQL EXECUTE
BEGIN
 ORDER_PROC(:s_ono in);
END;
END-EXEC;
```

[Example 2] The following is an example of executing stored function ORDER_FUNC.

```
< Example Program : psm2.sc >
EXEC SQL BEGIN DECLARE SECTION;
long long s_ono;
char s_order_date[19+1];
int s_eno;
char s_cno[13+1];
char s_gno[10+1];
int s_qty;
int s_cnt;
EXEC SQL END DECLARE SECTION;
s_ono = 200000001;
s_eno = 20;
s_qty = 2300;
```

```
strcpy(s_order_date, "19-May-03");
strcpy(s_cno, "7111111431202");
strcpy(s_gno, "C111100001");
EXEC SQL EXECUTE
BEGIN
 :s_cnt := ORDER_FUNC(:s_ono in, :s_order_date in,
 :s_eno in, :s_cno in,
:s_gno in, :s_qty in);
END;
END-EXEC;
```

Using Array Types Host Variables in EXECUTE Statements

The array-type host variable can be used in EXECUTE statement. By executing EXECUTE statement once using the array-type host variable, the user can get same effects as if executing EXECUTE statement as many time as the array size and expect the performance improvement.

Array Types

In EXECUTE statement, the array type can be used only for IN type.

The following shows the array types that can be used in EXECUTE statement.

Numeric-type or character-type arrays

Array type of the structure element

Restrictions

Some restrictions exist in relation to the use of array-type host variable in EXECUTE statement. Note the following when writing a program:

As the array component cannot be defined in the embedded SQL statement, the array type of the structure cannot be used.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
struct tag1 { int i1; int i2; int i3; } var1[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE BEGIN
PROC1(:var1[0].i1 in,
:var1[0].i2 in,
:var1[0].i3 in); (X)
END;
END-EXEC;
```

OUT or IN OUT type parameter cannot use the array-type host variable.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1[10];
int var2[10];
int var3[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE BEGIN
PROC1(:var1 in,
:var2 out,
:var3 in out); (X)
END;
END-EXEC;
```

The return type of the stored function is OUT type so that the return type cannot be an array.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1[10];
int var2[10];
int var3[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE BEGIN
:var1 = FUNC1(:var2 in, :var3 in); (X)
END;
END-EXEC;
```

The array type must not be used together with a non-array type.

```
Example) EXEC SQL BEGIN DECLARE SECTION;
int var1;
int var2;
int var3[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE BEGIN
PROC1(:var1 in, :var2 in, :var3 in); (X)
END;
END-EXEC;
```

Due to the third and fourth limitations in the above, the array-type host variable cannot be used in the execution statement of the stored function.

Example

In the following example, the array type is used as the host variable of IN type of the stored procedure statement.

```
< Example Program : arrays2.sc >
EXEC SQL BEGIN DECLARE SECTION;
char a_gno[3][10+1];
char a_gname[3][20+1];
EXEC SQL END DECLARE SECTION;
strcpy(a_gno[0], "G111100001");
strcpy(a_gno[1], "G111100002");
strcpy(a_gno[2], "G111100003");
strcpy(a_gname[0], "AG-100");
strcpy(a_gname[1], "AG-200");
strcpy(a_gname[2], "AG-300");
EXEC SQL EXECUTE
BEGIN
 GOODS_PROC(:a_gno in, :a_gname in);
END;
END-EXEC;
```

Sample Programs

psm1.sc

See $ALTIBASE_HOME/sample/APRE/psm1.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make psm1
shell> ./psm1
<SQL/PSM 1>
-----------------------------------------------------
[Create Procedure]
-----------------------------------------------------
Success create procedure
-----------------------------------------------------
[Execute Procedure]
-----------------------------------------------------
Success execute procedure
-----------------------------------------------------
[Drop Procedure]
-----------------------------------------------------
Success drop procedure
```

psm2.sc

See $ALTIBASE_HOME/sample/APRE/psm2.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make psm2
shell> ./psm2
<SQL/PSM 2>
-----------------------------------------------------
[Create Function]
-----------------------------------------------------
Success create function
-----------------------------------------------------
[Execute Function]
-----------------------------------------------------
31 rows selected
-----------------------------------------------------
[Drop Function]
-----------------------------------------------------
Success drop function
```

13 Multi-Connection Program

Overview

Allow one or more connections within one program of the embedded SQL statement. The following describes the definition and use of the multi-connection program.

Definition

A multi-connection program means a program that uses more than one connections.

The Need for Multi-Connection

Uses of the multi-connection program

When the program needs to access multiple database servers When the program must enable multiple users to access the database server

In case of a multi-threaded program

For more information about the multi-threaded program, see Chapter XII.

Connection Name.

In the multi-connection program, each connection is identified by the connection name. The connection name must be unique in the program. Only one connection is allowed that does not use the connection name, and this is the default connection. The connection name must be defined upon establishment of the connection with the database server. The embedded SQL statement must be executed by the connection with this name. In case a connection is established with an existing name, "Already connected" error message will be displayed.

Syntax

The syntaxes using the connection name are as follows:

EXEC SQL [AT <conn_name | :conn_name>] …

Arguments

Both the string and the variable can be used as a connection name. In case the variable is used, this variable does not need to be declared in the DECLARE section of the host variable.

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

Program Code Steps

The multi-connection program writing order is not quite different from the order of writing a normal program. The following describes the order or wiring a multi-connection program.

Connect to the database server. Define the connection name.

Execute the embedded SQL statement using the name of the established connection.

Disconnect using the names of all established connections.

SQL Statements for Multi-Connections

The using method of the embedded SQL statement in the multi-connection program is not greatly different form the using method of general embedded SQL statement. The basic syntax is same. Define the connection name using AT clause. The following describes how to use the embedded SQL statement in the multi-connection program.

CONNECT

Connect to the database server with the defined connection name.

Syntax

```
EXEC SQL [ AT <conn_name | :conn_name> ]

CONNECT <:user> IDENTIFIED BY <:passwd>

[ USING <:conn_opt1> [ , <:conn_opt2> ] ];
```

Arguments

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<:user> : User name to connect to the database server.

<:passwd> : User password to connect to the database server.

<:conn_opt1> : See Chapter VII.

<:conn_opt2> : See Chapter VII.

Description

When establishing more than one connections in one program, the connection name must be defined. The connection name must be unique in the program. Specify the connection name to use using AT clause in the embedded SQL statement.

Precautions

In the multi-connection program, only one connection that does not have name is allowed. The embedded SQL statement that does not use AT clause must be processed by this connection.

In case the user attempts to establish a connection with the same connection name, "Already connected" error message will be displayed. To establish a connection using the same name, execute FREE or DISCONNECT first. At this time, if the database server is running, execute DISCONNECT. Otherwise, perform FREE.

Examples

Displays various examples of establishing connections using the connection names.

[Example 1] In the following example, the database server is connected by the connection name using the string. "CONN1" will be the connection name.

```
< Example Program : mc1.sc >
EXEC SQL BEGIN DECLARE SECTION;
char usr[10];
char pwd[10];
EXEC SQL END DECLARE SECTION;
/* set username */
strcpy(usr, "SYS");
/* set password */
strcpy(pwd, "MANAGER");
/* connect to ALTIBASE server with CONN1 */
EXEC SQL AT CONN1 CONNECT :usr IDENTIFIED BY :pwd;
```

[Example 2] In the following example, the database server is connected by the connection name using the host variable. "CONN2" will be the connection name.

```
< Example Program : mc2.sc >
char usr[10];
char pwd[10];
char conn_name2[10];
/* set username */
strcpy(usr, "ALTITEST");
/* set password */
strcpy(pwd, "ALTITEST");
/* set connname */
strcpy(conn_name2, "CONN2");
/* connect to ALTIBASE server with :conn_name2 */
EXEC SQL AT :conn_name2 CONNECT :usr IDENTIFIED BY :pwd;
```

DISCONNECT

The database server will be disconnected by the defined connection name.

Syntax

EXEC SQL [AT <conn_name | :conn_name>]

DISCONNECT;

Argument

<conn_name> : Connection name. Conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

Description

The connection name should be valied, namely, pre-connected name.

In the multi-connection program, all connections must be disconnected by the connection name.

Examples

Displays various examples of disconnections using the connection name.

[Example 1] In the following example, the database server is disconnected by the connection name using the string. "CONN1" will be disconnected.

```
< Example Program : mc1.sc >
EXEC SQL AT CONN1 DISCONNECT;
```

[Example 2] In the following example, the database server is disconnected by the connection name using the host variable. "CONN2" will be disconnected.

```
< Example Program : mc2.sc >
char conn_name2[10];
strcpy(conn_name2, "CONN2");
EXEC SQL AT :conn_name2 DISCONNECT;
```

The Basic SQL Statements

Can execute the DML statement such as SELECT, UPDATE and the DDL statement such as CREATE, DROP .

Syntax

EXEC SQL [AT <conn_name | :conn_name>]

[SELECT | UPDATE | INSERT | DELETE |

CREATE] …

Argument

<conn_name> : Connection name. Conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

Description

The connection name should be valied, namely, pre-connected name.

Cursor Statements

Can executes the cursor-related embedded SQL statement.

Syntax

EXEC SQL [AT <conn_name | :conn_name>]

[DELCARE | OPEN | FETCH | CLOSE]

<cursor_name> …

Arguments

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<cursor_name> : Cursor name

Description

The connection name should be valid, namely, pre-connected name.

Dynamic SQL Statements

Can execute dynamic SQL statements.

Syntax

- Method 1

EXEC SQL [AT <conn_name | :conn_name>]

EXECUTE IMMEDIATE …

- Method 2

EXEC SQL [AT <conn_name | :conn_name>] PREPARE …

EXEC SQL [AT <conn_name | :conn_name>] EXECUTE …

- Method 3

EXEC SQL [AT <conn_name | :conn_name>] PREPARE …

EXEC SQL [AT <conn_name | :conn_name>] DECLARE …

EXEC SQL [AT <conn_name | :conn_name>] OPEN …

EXEC SQL [AT <conn_name | :conn_name>] FETCH …

EXEC SQL [AT <conn_name | :conn_name>] CLOSE …

Argument

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

Description

The connection name should be valid, namely, pre-connected name.

Other SQL statements

Can execute other embedded SQL statements.

Syntax

EXEC SQL [AT <conn_name | :conn_name>]

AUTOCOMMIT { ON | OFF };

EXEC SQL [AT <conn_name | :conn_name>]

COMMIT;

EXEC SQL [AT <conn_name | :conn_name>]

SAVEPOINT <savepoint_name>;

EXEC SQL [AT <conn_name | :conn_name>]

ROLLBACK [TO SAVEPOINT <savepoint_name>];

EXEC SQL [AT <conn_name | :conn_name>]

FREE;

EXEC SQL [AT <conn_name | :conn_name>]

BATCH;

Argument

<conn_name> : Connection name. Conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<savepoint_name> : Name of the storing point

Description

The connection name should be valid, namely, pre-connected name.

Exception

In the multi-connection program, following embedded SQL statement do not use AT clause:

Syntax

EXEC SQL INCLUDE …

EXEC SQL OPTION …

EXEX SQL WHENEVER …

Argument

None

Description

Note that the above three embedded SQL statements do not use AT clause when writing a program.

Stored Procedures in the Multi-connection Program

The stored procedure-processing SQL statement can be used in multi-connection program. Same as the syntax described above except that AT clause is added.

CREATE

Can create a stored procedure or stored function in the multi-connection program.

Syntax

```
EXEC SQL [ AT <conn_name | :conn_name> ]

CREATE [ OR REPLACE ] <PROCEDURE | FUNCTION>

…

END

[ <procedure_name | function_name> ] ;

END-EXEC;
```

Arguments

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

Description

The connection name should be valid, namely, pre-connected name.

ALTER

Can recompile a stored procedure or stored function in the multi-connection program.

Syntax

```
EXEC SQL [ AT <conn_name | :conn_name> ]

ALTER PROCEDURE

<procedure_name | function_name> COMPILE;
```

Argument

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

Description

The connection name should be valid, namely, pre-connected name.

DROP

Can create a stored procedure or stored function in the multi-connection program.

Syntax

EXEC SQL [AT <conn_name | :conn_name>] DROP

PROCEDURE <procedure_name | function_name>;

Argument

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

Description

The connection name should be valid, namely, pre-connected name.

EXECUTE

Can create a stored procedure or stored function in the multi-connection program.

Syntax

EXEC SQL [AT <conn_name | :conn_name>]

EXECUTE BEGIN

```
/* PL/SQL block here */

END;

END-EXEC;
```

Argument

<conn_name> : Connection name. conn_name will become the connection name.

<:conn_name> : Connection name. The string stored in conn_name will become the connection name.

<procedure_name> : Name of the stored procedure

<function_name> : Name of the stored function

/* Stored procedure block here */ : See Stored Procedure User's Manual

Description

The connection name should be valid, namely, pre-connected name.

Sample Programs

mc1.sc

See $ALTIBASE_HOME/sample/APRE/mc1.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make mc1
shell> ./mc1
<MULTI CONNECTION 1>
----------------------------------------------------
[Declare Cursor With CONN1 ]
----------------------------------------------------
Success declare cursor with CONN1
----------------------------------------------------
[Open Cursor With CONN1]
----------------------------------------------------
Success open cursor with CONN1
----------------------------------------------------
[Fetch Cursor With CONN1 -> Insert With CONN2]
----------------------------------------------------
30 rows inserted
----------------------------------------------------
[Close Cursor With CONN1]
----------------------------------------------------
Success close cursor with CONN1
```

mc2.sc

See $ALTIBASE_HOME/sample/APRE/mc2.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make mc2
shell> ./mc2
<MULTI CONNECTION 2>
----------------------------------------------------
[Dynamic SQL Method 1 With :conn_name2]
----------------------------------------------------
Success dynamic sql method 1 with :conn_name2
----------------------------------------------------
[Dynamic SQL Method 2 (PREPARE) With :conn_name2]
----------------------------------------------------
Success dynamic sql method 2 (prepare) with :conn_name2
----------------------------------------------------
[Dynamic SQL Method 3 (PREPARE) With :conn_name1]
----------------------------------------------------
Success dynamic sql method 3 (prepare) with :conn_name1
----------------------------------------------------
[Dynamic SQL Method 3 (DECLARE CURSOR) With :conn_name1]
----------------------------------------------------
Success dynamic sql method 3 (declare cursor) with :conn_name1
----------------------------------------------------
```

```
[Dynamic SQL Method 3 (OPEN CURSOR) With :conn_name1]
----------------------------------------------------
Success dynamic sql method 3 (open cursor) with :conn_name1
----------------------------------------------------
[Dynamic SQL Method 3 (FETCH CURSOR) With :conn_name1
 -> Dynamic SQL Method 2 (EXECUTE-INSERT) With :conn_name2]
----------------------------------------------------
20 rows inserted
----------------------------------------------------
[Dynamic SQL Method 3 (CLOSE CURSOR) With :conn_name1]
----------------------------------------------------
Success dynamic sql method 3 (close cursor) with :conn_name1
```

mc3.sc

See $ALTIBASE_HOME/sample/APRE/mc3.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make mc3
shell> ./mc3
<MULTI CONNECTION 3>
----------------------------------------------------
[Autocommit Off With CONN1]
----------------------------------------------------
Autocommit mode of CONN1 session modified false
----------------------------------------------------
[Autocommit Off With CONN2]
----------------------------------------------------
Autocommit mode of CONN2 session modified false
----------------------------------------------------
[Create Procedure With CONN1]
----------------------------------------------------
Success create procedure
----------------------------------------------------
[Create Procedure With CONN2]
----------------------------------------------------
Success create procedure
----------------------------------------------------
[Execute Procedure With CONN1]
----------------------------------------------------
Success execute procedure
----------------------------------------------------
[Execute Procedure With CONN2]
----------------------------------------------------
Success execute procedure
----------------------------------------------------
[Commit With CONN1]
----------------------------------------------------
Success commit
----------------------------------------------------
[Commit With CONN2]
----------------------------------------------------
Success commit
----------------------------------------------------
[Drop Procedure With CONN1]
----------------------------------------------------
Success drop procedure
-----------------------------------------------------------
[Drop Procedure With CONN2]
```

```
----------------------------------------------------
Success drop procedure
```

14 Multi-threaded Program

Multi-threaded Program

The embedded SQL statement supports the multi-threaded program. The following describes how to use the embedded SQL statement in the multi-threaded program and things to note when using it:

Configuration

Upon precompiling, the judgment background for the multi-threaded program must be provided for the precompiler in one of following two ways:

Setting -mt option in the command line

< Example >

```
shell> apre -mt sample1.sc
```

Using OPTION statement in the file

< Example >

```
EXEC SQL OPTION (THREADS = TRUE);
```

Description

- Each thread must have its own connection. In other words, multiple threads must not share one connection.
- The connection name must be unique in the program. In case the user attempts to establish a connection with the same name, "Already connected" error message will be displayed.
- Each connection must have its own connection name. Only one connection without a connection name must be allowed.
- The embedded SQL statement must indicate the connection name. For more information about using the connection name, see Chapter XI.
- All threads must call ideAllocErrorSpace() function. This function must be called before the embedded SQL statement is used.
- When sending sqlca using the function argument, send it as ssqlca.

* Notes: When the C/C++ precompiler version is 3.5.5 or lower, EXEC SQL THREADS; must be defined in all functions using the embedded SQL statement. In the later versions, this does not need to be defined because EXEC SQL THREADS syntax in the existing program is compatible.

Sample Programs

mt1.sc

See $ALTIBASE_HOME/sample/APRE/mt1.sc

Execution Result

```
shell> is -f schema/schema.sql
shell> make mt1
shell> ./mt1
<MULTI THREADS 1>
```

mt2.sc

See $ALTIBASE_HOME/sample/APRE/mt2.sc

Execution Result

```
shell> is –f schema/schema.sql
shell> make mt2
shell> ./mt2
<MULTI THREADS 2>
----------------------------------------------------
ORDER_DATE ENO GNO
----------------------------------------------------
2000/11/29 00:00:00 12 A111100002
2000/11/29 00:00:00 12 E111100001
2000/11/29 00:00:00 19 E111100001
2000/12/10 00:00:00 19 D111100008
2000/12/01 00:00:00 19 D111100004
2000/12/29 00:00:00 12 C111100001
2000/12/29 00:00:00 20 E111100002
2000/12/30 00:00:00 20 D111100002
2000/12/30 00:00:00 19 D111100008
2000/12/30 00:00:00 20 A111100002
2000/12/30 00:00:00 12 D111100002
2000/12/30 00:00:00 20 D111100011
2000/12/30 00:00:00 20 D111100003
2000/12/30 00:00:00 19 D111100010
2000/12/30 00:00:00 20 C111100001
2000/12/30 00:00:00 12 E111100012
2000/12/30 00:00:00 20 C111100001
2000/12/30 00:00:00 12 F111100001
```

15 Error Code/Messages

Error Code/Messages

Error codes occurring during precompiling are as follows:

ERR-xxxY

xxx : Error number

Y : Error classification

Error Numbers

Depending on the error type, the error numbers are classified as follows:

101 – 199 : System error

201 – 299 : Host variable-related error

301 – 399 : Cursor-related error

401 – 499 : General error

701 – 799 : Features that the current version does not support

Error Classification

Error classification means the precompiling procedure where the error occurred.

E : When the error occurs in the DECLARE section of the host variable

L : When the error occurs in the processing part of the embedded SQL statement

M: This denotes that an error is raised in case of performing the required substitutions.

H: When the error occurs in other parts than the above

Error Message Format

[ERR-<error code> : <error message>]

line <line number>: <error statement>

<error code> : Error code

<error message> : Error message

<line number> : Row number where the error occurred

<error statement>: Statement where the error occurred

Example

The following example is an error displayed when an incorrect type is used in declaration of the host variable.

```
[ERR-302L : EXEC SQL END DECLARE SECTION is not exist.]
File : tmp.sc
Line : 4
Offset: 1-31
Error_token:EXEC SQL BEGIN DECLARE SECTION;
```

Error Code/Messages

101H – 199H

Error Code	Error Message
101H	File '<0%s>' open error
102H	FileSize(<0%s>) is zero.
103H	The include file [<0%s>] does not exist.
104H	File '<0%s>' delete error
105H	Memory allocation error.
106H	Latch initialization error. (<1%s>:<0%d>)
107H	Latch read error. (<1%s>:<0%d>)
108H	Latch release error. (<1%s>:<0%d>)
109H	Latch write error. (<1%s>:<0%d>)
110H	Latch destroy error. (<1%s>:<0%d>)
111H	File close error.
112H	File <0%s> write error

201E – 299E

Error Code	Error Message
201E	C-type comment is not closed.
202E	The structure name [<0%s>] is unknown.

Error Code	Error Message
203E	The structure name [<0%s>] is a duplicate.
204E	The symbol name [<0%s>] cannot be added to the symbol table.
205E	The symbol name exceeds the maximum possible length. [<0%s>]
206E	redefinition of '<0%s>'
207E	unknown type of '<0%s>'
208E	The scope depth defined by the braces is too high [<0%s>].
209E	Brace count error
210E	Parenthesis count error
211E	The nested structure exceeds the maximum possible depth.

301L – 399L

Error Code	Error Message
301L	The C include file cannot contain embedded SQL statements.
302L	EXEC SQL END DECLARE SECTION does not exist.
303L	EXEC SQL BEGIN DECLARE SECTION does not exist.
304L	EXEC SQL END ARGUMENT SECTION does not exist.
305L	EXEC SQL BEGIN ARGUMENT SECTION does not exist.
306L	Unterminated string error.
307L	The connection name[<0%s>] is too long. (The maximum length is 50 characters.)
308L	The cursor name[<0%s>] is too long. (The maximum length is 50 characters.)
309L	Statement name[<0%s>] is too long. (The maximum length is 50 characters.)
310L	The number of FOR loop iterations must be greater than zero.
311L	The host variable[<0%s>] is unknown.
312L	The host variable in a FREE LOB statement must be a LOB locator.
313L	Unterminated embedded SQL statement.
314L	The indicator variable [<0%s>] should be of type SQLLEN or a compatible type.
315L	Two or more arrays of structures are bound to host variables in the same statement.

401M – 499M

Error Code	Error Message
401M	An unknown macro is too long. (>2k)
402M	Macro #if statement syntax error
403M	Macro #elif statement syntax error
404M	Macro #elif statement sequence error
405M	Macro #else statement sequence error
406M	Macro #endif statement sequence error
407M	An empty char constant cannot be used in a macro without an #endif statement.
408M	Include files are nested too deeply. (maximum <0%s>)
409M	No #endif error.
410M	A closing parenthesis ‘)’ is missing from the macro parameter list.
411M	Unknown macro name, or missing parenthesis after macro name. (<0%s>)
412M	Unterminated string error

501H – 599H

Error Code	Error Message
501H	The <0%s> option was repeatedly used.
502H	An option string <0%s> was too long.
503H	The -mt and -sea options cannot be used together.
504H	The names of Input file must be in the form of '*.sc'.
505H	Unknown embedded SQL statement type.

701L – 799L

Error Code	Error Message
701L	The CURSOR SENSITIVITY option is not supported yet.
702L	Any CURSOR INSENSITIVITY option does not supported, yet.
703L	Any CURSOR ASENSITIVITY option does not supported, yet.
704L	WITH HOLD option does not supported, yet.
705L	WITH RETURN option does not supported, yet.
706L	READ ONLY option does not supported, yet.
707L	ALTER COMPACT option does not supported, yet.
708L	The host variable of array of structure type can't be repeated.

Appendix A. Coding Guide

Coding Guide

There are some considerations on writing an embedded SQL statement program. The developer is recommended to read recommendations and notes in the appendix to minimize mistakes and to maximize the efficiency of programming work.

About Host Variables

Note related to the host variables

- The nested structure cannot be used as a host variable. In other words, the element of the structure cannot be a structure.
- The host variable must not be a pointer type. In exceptional cases, char* is allowed.
- Macro can be used only to define the number of elements to be listed upon declaration of the array-type host variables. For example, macro definition cannot be used in the place where the host variable can be used in the internal SQL statement.
- The character type (char, varchar). The host variable must be defined higher than the corresponding column size by 1. Otherwise, the column value will be cut after the execution of SELECT statement or FETCH statement and sqlca.sqlcode will be SQL_SUCCESS_WITH_INFO.

About Indicator Variables

Notes on related to the indicator variable

- The datatype of the indicator variable must be 'int' type.
- In case a separate indicator variable is not defined when "varchar" type is used as the input host variable, define len. If the array is not NULL, set the array length as the len. If the array is NULL, set -1 as the len.
- If the indicator variable is not -1 when the host variable is the numeric datatype, the indicator variable does not have any meaning.
- If the host variable type is the binary datatype, the indicator variable must be used.

About DECLARE section of the Host Variables

Notes related to the DECLARE section of the host variable

- In case of a definition of the structure data, the data must be defined after the structure is defined. Or both the structure and the data must be defined at the same time. In case the structure is defined after the data is defined, using the host variable of this data will cause an error.

- In the host variable declaration part, the user cannot set the value. In other words, initialization of the value and declaration of the variable cannot be done simultaneously.
- Only the constant macro definition is allowed in the DECLARE section of the host variable.
- The definition of the datatype (typedef) to be used as the datatype of the host variable must be defined in DECLARE section of the host variable.

About Embedded SQL Statements

Notes related to the embedded SQL statement

- When the host variable is a structure, the indicator variable must be a structure. At this time, two structures must have the same number of elements.
- If the number of results returned after execution of SELECT statement is more one case or more than the array size, "Returns too many rows" error message will be displayed.

About Host Array Variables

Notes on array host variable

- the array host variable allows only one-dimensional array. In some cases, two-dimensional arrays for 'char' type and 'varchar' type are allowed.
- If the host variable is an array of the structure, the indicator variable cannot be used.
- In case the structure array is used as a host variable in INTO clause of SELECT or FETCH statement, only one output host variable can be used. In other words, it must not be used with other host variables. Therefore, if the host variable to be used in INTO clause is a structure, the number of components must be the same as the number of the columns in SELECT clause.
- In case the structure array is used as a host variable in VALUES clause of INSERT statement, only one input host variable must be used. In other words, it must not be used with other host variables. Therefore, if the host variable to be used in VALUES clause is an array type of the structure, the number of the elements of this structure must be the same as the number of columns in INSERT statement.
- 'varchar' type is a structure internally so it is subject to above limitations.
- The array type must not be used together with a non-array type.
- If an array-type output host variable is used upon execution of SELECT statement or FETCH statement and the number or the returned records is smaller than the array size, sqlca.sqlcode will be SQL_SUCCESS.
- While the SELECT statement or CURSOR statement is executed, the input host variable cannot be an array.
- In UPDATE statement, DELETE statement, and stored procedure statement, an array-type host variable of the structure cannot be used. The reason is that it is not possible to define array ele-

ments in the embedded SQL statement.

- When the input host variable is an array, FOR clause can be used in INSERT statement, UPDATE statement, and DELETE statement.
- When using an array-type host variable in AUTOCOMMIT mode, each array element, not an entire array, is one transaction. Therefore, if only some elements are successful while others are not, the changes in the successful transaction are permanently stored in the database server.
- The pointer type cannot be declared by the array.
- In the embedded SQL statement, the array component cannot be defined.

About Connect and Session

The following describes things to note in relation with CONNECT statement, multi-connection, and the SESSION.

- Before establishing a connection using an existing connection name, execute FREE statement or DISCONNECT statement. If the database server is running, execute DISCONNECT statement. Otherwise, execute FREE statement.
- When setting CONNTYPE as 2 or 3 while defining the connection method with USING clause, DSN and PORT_NO options will be ignored although they are set. Instead, the connection with the database server will be attempted.
- Assume that there are two connection options. If the connection is established with the first option, sqlca.sqlcode will be SQL_SUCCESS. If the connection is established with the second option, sqlca.sqlcode will be SQL_SUCCESS_WITH_INFO. However, if the connection is not established by any option, sqlca.sqlcode will be SQL_ERROR.
- Maximum 1024 embedded SQL statements are allowed per connection.
- In case program is terminated without committing is made in AUTOCOMMIT OFF session, all SQL statement not committed will be rolled back. However, if the program is terminated after DISCONNECT statement is executed, all executed embedded SQL statement will be committed.
- In the following embedded SQL statement, AT clause is not allowed.
- INCLUDE statement : EXEC SQL INCLUDE …
- OPTION Statement : EXEC SQL OPTION …
- WHENEVER statement: EXEC SQL WHENEVER …

About Runtime Error Handling

The following describes note on using SQLCA, SQLCODE, SQLSTATE and WHENEVER statement such as execution time error handling.

- After executing every embedded SQL statement, check sqlca.sqlcode for correct error handling.

- If the output host variable size is smaller than the corresponding column size in SELECT statement, the data will be cut to be saved in the host variable. At this time, sqlca.sqlcode will be SQL_SUCCESS_WITH_INFO.

- If no record is affected by Update or Delete operation, sqlca.sqlcode will be SQL_SUCCESS. To check the number of records that are affected by Update or Delete operation, see sqlca.sqlerrd[2]. In the above example, this value is 0.

- In the SQLCODE, the error code is a negative value. However, in Error Message Reference , the error code is a positive hexadecimal value. Therefore, when referring to Error Message Reference, convert the absolute value of the SQLCODE into a hexadecimal data.

- The application scope of WHENEVER statement is different from the program flow, and it is valid only in the current file.

- WHENEVER statement must be declared before the embedded SQL statement to be applied.

- WHENEVER statement is independent of the connection. In other word, declaration of WHENEVER statement in a file with one or more connections, all embedded SQL statements within the corresponding range will be affected regardless of the connection.

About Multi-threaded Programming

Notes related to the multi-threaded program

- Each thread must have its own connection. In other words, multiple threads must not share one connection.

- Each connection must have its own connection name. Only one connection without a connection name must be allowed.

- The embedded SQL statement must indicate the connection name.

- All threads must call ideAllocErrorSpace() function. This function must be called before the embedded SQL statement is used.

About Naming Conventions

Naming rules for programming

- The host variable name and the indicator variable name must start with the alphabet letter (a ~ z, A ~ Z) and (_) and the length must not exceed 50 letters.

- The cursor name must start with the alphabet letter (a ~ z, A ~ Z) and (_) and the length must not exceed 50 letters.

- The SQL statement identifier of the dynamic SQL statement must start with the alphabet letter (a ~ z, A ~ Z) and (_) and the length must not exceed 50 letters.

- The connection name must start with the alphabet letter (a ~ z, A ~ Z) and (_) and the length must not exceed 50 letters.

- The maximum length of SQL statement is 32Kbytes.

Others

The following describes other things that the programmer must take note of.

- The header file cannot be mutually referred to. In other words, myheader1.h must not refer to myheader2.h and myheader2.h must not refer to myheader1.h.

Appendix B. Conversion between Pro*C and C/C++ Precompiler

이 부록은 오라클의 pro*C(C++) 로 작성된 응용 프로그램을 알티베이스 C/C++ Precompiler 응용 프로그램으로 전환할 때 참조한다 .

Datatypes

This section describes about Oracle datatype and the corresponding ALTIBASE datatype.

Datatype Comparison Table

ODBC SQL datatype	Oracle	Altibase	Comments
SQL_CHAR	CHAR	CHAR	1-255 length
SQL_TIMESTAMP	DATE	DATE	1-4000 length
SQL_LONGVARCHAR	LONG	BLOB	Up to 32K
SQL_INTEGER	INT	INTEGER	
SQL_FLOAT	NUMBER	NUMBER	
SQL_DECIMAL	NUMBER(P)	NUMBER(P)	1-38
SQL_DECIMAL	NUMBER(P,S)	NUMBER(P,S)	precision : 1-38 scale : -84 – 126
SQL_BINARY	RAW	HSS_BYTES	1-255
SQL_VARCHAR	VARCHAR	VARCHAR	max 32K
SQL_VARCHAR	VARCHAR2	VARCHAR	max 32K

Embedded Functions

Like Oracle DBMS, ALTIBASE provides number functions, date functions, string functions, datatype conversion functions and buit-in functions. This section describes ALTIBASE datatype conversion functions and build-in functions that correspond to the Oracle.

Comparison of Embedded Functions

- Function name, usage and methods are similar to Oracle.
- Function types supported by ALTIBASE.

Number functions: ABS, ACOS, ASIN, ATAN, ATAN2, CEIL, COS, COSH, EXP, FLOOR, etc.

Group functions: COUNT, MAX, AVG, MIN, etc.

String functions: ASCII, CHR, CONCAT, CHAR_LENGTH, INSTR, LOWER, UPPER, LTRIM, etc.

Date functions : ADD_MONTHS, EXTRACT, LAST_DAY, NEXT_DAY, SYSDATE, etc.

Data conversion functions: TO_CHAR, TO_NUMBER, TO_DATE

Miscellaneous functions: DECODE, NVL, CASE2, GREATEST, etc.

For more information, see the SQL User's Manual.

The following highlights comparison of embedded functions between Oracle and Altibase, which are frequently used by the application program.

Usage	Oracle	Altibase
Character to number	TO_NUMBER(expression)	TO_NUMBER(expression)
Number to character	TO_CHAR(expression)	TO_CHAR(expression)

Character to date	TO_DATE('04-JUL-97')
	TO_DATE('04-JUL-1997')
	TO_DATE('04-JUL-1997', 'dd-mon-yyyy')
		TO_DATE('04-JUL-97')
		TO_DATE('04-JUL-1997')
		TO_DATE('04-JUL-1997', 'dd-mon-yyyy')

Date to character	TO_CHAR(expression)
	TO_CHAR(expression, 'dd mon yyyy')
	TO_CHAR(expression, 'mm/dd/yyyy')
		TO_CHAR(expression)
		TO_CHAR(expression, 'dd mon yyyy')
		TO_CHAR(expression, 'mm/dd/yyyy')

Conditional Value Selection	DECODE(expression, search1, result1, [search2, result2,….] [,default])	DECODE(expression, search1, result1, [search2, result2,….] [,default])

Database Connection

This chapter describes different database connection/disconnection methods between Oracle and Altibase.

Database Connection

Oracle and Altibase use the same syntax for default connections. Once the connection name has been assigned, multiple connections are available. Like Oracle, connection options can be set using the USING clause.

Connect Statement

- Oracle

```
EXEC SQL CONNECT {:user IDENTIFIED BY :oldpswd :usr_psw }
[[ AT { dbname | :host_variable }] USING :connect_string ];
```

- Altibase

```
EXEC SQL CONNECT <:user> IDENTIFIED BY <:passwd> [USING
<:conn_opt>[,<:conn_opt2>]];
```

Example of Default Connection

- Oracle

```
char *username = "SCOTT";
char *password = "TIGER";
char *connstr = "ORA817";
EXEC SQL WHENEVER SQLERROR
.
.
.
EXEC SQL CONNECT :username IDENTIFIED BY :password
USING :connstr;
```

- Altibase

```
strcpy(username, "SYS");
strcpy(password, "MANAGER");
strcpy(connstr,"DSN=192.168.1.2;PORTNO=20310;CONNTYPE=3");
EXEC SQL CONNECT :username IDENTIFIED BY :password
USING :connstr;
```

If you do not specify the “USING” clause, connection is established to the ALTIBASE in the system that runs the application program.

Example of Explicit Connection

- Oracle

```
char *username = "SCOTT";
char *password = "TIGER";..
EXEC SQL WHENEVER SQLERROR
```

```
.
.
.
EXEC SQL CONNECT :username
IDENTIFIED BY :password;
```

- ALTIBASE

```
strcpy(user2, "ALTIBASE");
strcpy(passwd2, "ALTIBASE");
strcpy(conn_name, "CONN2");
EXEC SQL AT :conn_name CONNECT :user2
IDENTIFIED BY :passwd2;
```

Database Disconnection

Unlike Oracle, Altibase does not support a statement (EXEC SQL ROLLBACK WORK RELEASE) that disconnects the database while performing rollback.

Execute EXEC SQL ROLLBACK; EXEC SQL DISCONNECT; continuously to get the same effect.

Disconnect Statement

- Oracle

```
EXEC SQL COMMIT WORK RELEASE;
```

or

```
EXEC SQL ROLLBACK WORK RELEASE;
```

- ALTIBASE

```
EXEC SQL DISCONNECT;
```

Host Variables

This section describes differences between host variables used by Oracle Pro*C and those used by Altibase C/C++ Precompiler.

Host Variable Compability

Oracle	C type	ALTIBASE	C type	Remarks
CHAR	Char	CHAR	char/char[2]	single character

VARCHAR2(X)
VARCHAR(X)
	VARCHAR[X]	VARCHAR	VARCHAR	n-byte variable-length character array

CHAR[X]	char[x]	CHAR[X]	char[x]	n-byte character array

Oracle	C type	ALTIBASE	C type	Remarks
NUMBER	Int	NUMBER/ INTEGER	int/APRE_INT	Integer
NUMBER(P,S)	short int long float double	NUMBER(P,S)	short int/ APRE_INTEGER long float double	small integer integer large integer float-point number double-precision floating-point number
DATE	char[n] varchar[n]	DATE	char[n] varchar[n]	n >= 20

Host Variable Declaration Section

ALTIBASE uses the same format of statements of a declaration section as Oracle. Altibase requires host variable declaration in the DECLARE SECTION clause.

- Oracle

```
EXEC SQL BEGIN DECLARE SECTION;
/* Host variable declaration */
EXEC SQL END DECLARE SECTION;
```

- ALTIBASE

```
EXEC SQL BEGIN DECLARE SECTION;
/* Host variable declaration */
EXEC SQL END DECLARE SECTION;
```

Using Embedded SQL Statements

This section describes basic SQL statements (SELECT, UPDATE, INSERT, DELETE), cursor control SQL statements and dynamic SQL statements for ALTIBASE C/C++ Precompiler and Oracle Pro*C.

Basic DML Statements

The way of using basic DML statements, including SELECT, INSERT, UPDATE, DELETE is the same as Oracle.

Cursor Control SQL Statements

Basic cursor declaration method is identical for both Oracle and Altibase. Unlike Oracle, however, Altibase does not support cursor variable declaration in the DECLARE SECTION and cursor variable usage.

Cursor Declaration

- Oracle

```
 EXEC SQL DECLARE cur_emp CURSOR FOR
 SELECT ename, job, sal
 FROM emp;
```

- ALTIBASE

```
EXEC SQL DECLARE cur_emp CURSOR FOR
 SELECT ename, job, sal
FROM emp;
```

Cursor Open and Fetch

Cursor open and fetch methods are identical for both Altibase and Oracle. Error code type and value differ between Oracle and Altibase, however. Therefore, error handling codes inside the Fetch statement need to be converted. You can use the same method as Oracle for the WHENEVER statement - EXEC SQL WHENEVER NOT FOUND DO BREAK; to handle runtime error.

- Oracle

```
EXEC SQL OPEN cur_emp;
if(sqlca.sqlcode != SQL_OK ) {
fprintf(stderr, "OPEN CSR ERROR%d\n",sqlca.sqlcode);
 close_db();
 exit(0);
}
for(;;)
{
EXEC SQL FETCH cur_emp
 INTO :emp_name, :job_title, :salary;
 switch(sqlca.sqlcode)
 {
 case 0:
 printf("emp_name : %s\n", emp_name);
 continue;
 case 1403: /* Not Found Data */
 break;
 default :
 fprintf(stderr, "FETCH CSR ERROR %d",sqlca.sqlcode);
 close_db();
 exit(0);
 }
}
```

- Altibase

```
EXEC SQL OPEN cur_emp;
if(sqlca.sqlcode != SQL_SUCCESS ) {
 fprintf(stderr, "OPEN CSR ERROR %d\n",sqlca.sqlcode);
 close_db();
 exit(0);
}
for(;;)
{
 EXEC SQL FETCH cur_emp INTO :emp_name, :job_title, :salary;
 switch(sqlca.sqlcode)
 {
 case SQL_SUCCESS:
```

```
 printf("emp_name : %s\n", emp_name);
 continue;
 case SQL_NO_DATA: /* Not Found Data */
 break;
 default :
 fprintf(stderr, "FETCH CSR ERROR %d",sqlca.sqlcode);
 close_db();
 exit(0);
 }
}
```

Cursor Close

Cursor Close statement functions same both in Oracle pro*C and in ALTIBASE Precompiler.

- Oracle

```
EXEC SQL CLOSE cur_emp;
```

- Altibase

```
EXEC SQL CLOSE cur_emp;
```

Dynamic SQL Statements

Altibase supports method 1, 2 and 3 but not method 4.

As the parameter marketer, Oracle uses v[1...n] while Altibase uses '?'.

Method 1

- Oracle

```
char dynstmt1[80];
strcpy(dynstmt1, "DROP TABLE EMP" );
EXEC SQL EXECUTE IMMEDIATE :dynstmt1;
```

- Altibase

```
EXEC SQL BEGIN DECLARE SECTION;
char dynstmt1[80];
EXEC SQL END DECLARE SECTION;
strcpy(dynstmt1, "DROP TABLE EMP" );
EXEC SQL EXECUTE IMMEDIATE :dynstmt1;
```

Method 2

- Oracle

```
int emp_number;
char delete_stmt[120];
.
.
.
strcpy(delete_stmt, "DELETE FROM EMP WHERE EMPNO = :v1");
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
emp_number = 10;
```

```
EXEC SQL EXECUTE sql_stmt USING :emp_number;
```

- Altibase

```
EXEC SQL BEGIN DECLARE SECTION;
int emp_number;
char delete_stmt[120];
EXEC SQL END DECLARE SECTION;
.
.
.
strcpy(delete_stmt, "DELETE FROM EMP WHERE EMPNO = ?");
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
emp_number = 10;
EXEC SQL EXECUTE sql_stmt USING :emp_number;
```

Method 3

- Oracle

```
char sql_query[80];
int deptno = 10;
char ename[10];
strcpy(sql_query,"SELECT ename FROM emp WHERE deptno > :v1");
EXEC SQL PREPARE S FROM :dynstmt;
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C USING :deptno;
for (;;)
{
 EXEC SQL FETCH C INTO :ename;
 .
 .
 .
}
```

- Altibase

```
EXEC SQL BEGIN DECLARE SECTION;
char sql_query[80];
int deptno = 10;
char ename[10];
EXEC SQL END DECLARE SECTION;
strcpy(sql_query,"SELECT ename FROM emp WHERE deptno > ? ");
EXEC SQL PREPARE S FROM :dynstmt;
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C USING :deptno;
for (;;)
{
 EXEC SQL FETCH C INTO :ename;
 .
 .
 .
}
```

Execution Returns and Status Codes

In this section, differences in SQLSTATE, SQLCODE and SQLCA values for handling runtime error will be highlighted for Oracle and Altibase.

SQLCA

SQLCA is a structure that saves execution results for embedded SQL statements. The members of the structure are sqlcode, sqlerrm.sqlerrmc, sqlerrm.sqlerrml and sqlerrd[2], which is used by ALTIBASE. Members that exist in only SQLCA of Oracle and not supported by ALTIBASE (e.g sqlwarn) cannot be used. sqlwarn)

SQLCA Declaration

- Oracle

```
EXEC SQL INCLUDE SQLCA;
or
#include <sqlca.h>
```

- Altibase

  Can be used without separate declaration.

sqlca.sqlcode status

- Oracle

Status Code	Description
0	Success
>0	No row returned
<0	database, system, network , application error

- Altibase

Status Code	Description
SQL_SUCCESS	Success
SQL_SUCCESS_WITH_INFO	
SQL_NO_DATA	No row returned
SQL_ERROR	
SQL_INVALID_HANDLE	

sqlca.sqlerrm

Usage and method for sqlerrmc and sqlerrml are identical for Oracle and Altibase.

sqlca.sqlerrd[2]

- Oracle

  The number of records affected by Insert/Update/Delete/Select Into operations (number of accumulated records)

- Altibase

  Number of records affected by Insert /Update /Delete operations

  Number of returned records when executing the SELECT or FETCH statement, if the output host variable is an array..

SQLSTATE

SQLSTATE saves the status code, which enables the user to review error and exception error.

Declaration and Usage of SQLSTATE

- Oracle

  Declare MODE=ANSI as the command string option of the precompiler and use it.

  ```
  Char SQLSTATE[6]
  ```

- Altibase

  No declaration is required.

Status Code of SQLSTATE

SQLSTATE status codes are different for Oracle and ALTIBASE in terms of meaning and value. Therefore, the user should perform conversions using the code table.

SQLCODE

SQLCODE saves error codes after executing the embedded SQL statement.

Declaration and Usage of SQLCODE

- Oracle

  Declare MODE=ANSI as the command string option of the precompiler and use it.

  ```
  long SQLCODE;
  ```

- Altibase

  No declaration is required.

  ```
  SQLCODE datatype in ALTIBASE is int.
  ```

Status Code Value of SQLCODE

- Oracle

  The same status code with sqlca.sqlcode is saved.

Status Code	Description
0	Upon successful execution of the embedded SQL statement When sqlca.sqlcode is SQL_SUCCESS:
1	When the embedded SQL statement is successfully executed but exception error occurs. When sqlca.sqlcode is SQL_SUCCESS_WITH_INFO:
100	When there is no record returned after SELECT or FETCH statement is executed: When sqlca.sqlcode is SQL_NO_DATA:
-1	When there is no corresponding error code for the error occurring during execution of an embedded SQL statement: At this time, sqlca.sqlcode is SQL_ERROR.
-2	When the embedded SQL statement is executed without database connection. That is, the sqlca.sqlcode value is SQL_INVALID_HANDLE.
If a value other than the values mentioned above is set to SQLCODE, it means that there is an error message due to an error in the respective SQL.	

Other Differences

This section illustrates differences in changing the commit mode, default commit mode and commit method between Altibase and Oracle.

Default Commit Mode

Oracle	ALTIBASE
non AUTOCOMMIT mode	auto commit mode

Changing the Commit Mode

- Oracle

```
EXEC SQL ALTER SESSION SET AUTOCOMMIT = TRUE or FALSE
```

- Altibase

```
EXEC SQL AUTOCOMMIT ON
```

```
or
EXEC SQL ALTER SESSION SET AUTOCOMMIT = TRUE or FALSE
```

Explicit Commit

- Oracle

```
EXEC SQL COMMIT;
or
EXEC SQL COMMIT WORK;
```

- Altibase

```
EXEC SQL COMMIT;
```

When Executing SELECT Statement in Non- AUTOCOMMIT Mode

- Oracle

  No need to commit the statement.

- Altibase

  Statement should be committed. EXEC SQL COMMIT; Execution)

Sample Programs

The following sample source contains the topics described above.

Oracle

```
#include <stdio.h>
#include <stdlib.h>
EXEC SQL include sqlca.h;
EXEC SQL BEGIN DECLARE SECTION;
char emp_name[21];
char job_title[21];
int salary;
int emp_number;
EXEC SQL END DECLARE SECTION;
char uid[10] = "SCOTT";
char pwd[10] = "TIGER";
int main(void)
{
 int dynamic_emp_number;
 char dynamic_stmt[120];
 EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
 if ( sqlca.sqlcode != 0 ) {
 fprintf(stderr, "DataBase Connect Error : [%d]!!!", sqlca.sqlcode);
 exit(-1);
 }
 /* INSERT */
 /* value setting */
 emp_number = 10;
```

```
strcpy(emp_name, "oracle1");
strcpy(job_title, "oracle dba1");
salary = 10000;
/* INSERT DML */
EXEC SQL INSERT INTO emp (empno, ename, job, sal)
VALUES (:emp_number, :emp_name, :job_title, :salary);
if ( sqlca.sqlcode != 0 ) {
fprintf(stderr, "DataBase Connect Error : [%d]!!!", sqlca.sqlcode);
exit(-1);
}
emp_number = 20;
strcpy(emp_name, "oracle2");
strcpy(job_title, "oracle dba2");
salary = 10000;
EXEC SQL INSERT INTO emp (empno, ename, job, sal)
VALUES (:emp_number, :emp_name, :job_title, :salary);
if ( sqlca.sqlcode != 0 ) {
fprintf(stderr, "Insert Error : [%d]!!!", sqlca.sqlcode);
exit(-1);
}
/* SELECT DML */
emp_number = 10;
EXEC SQL SELECT ename, job, sal INTO :emp_name, :job_title, :salary
FROM emp
WHERE empno = :emp_number;
if ( sqlca.sqlcode != 0 ) {
fprintf(stderr, "Select Error : [%d]!!!", sqlca.sqlcode);
exit(-1);
}
printf(" SELECT result : ename=[%s], job=[%s], sal=[%d]\n",
emp_name, job_title, salary);
/* UPDATE DML */
emp_number = 10;
salary = 2000;
EXEC SQL UPDATE emp SET sal = :salary WHERE empno = :emp_number;
if ( sqlca.sqlcode != 0 ) {
fprintf(stderr, "Update Error : [%d]!!!", sqlca.sqlcode);
exit(-1);
}
/* Cursor Create */
EXEC SQL DECLARE cur_emp CURSOR FOR
SELECT ename, job, sal FROM emp;

/* Cursor Open */
EXEC SQL OPEN cur_emp;
if(sqlca.sqlcode != 0 ) {
fprintf(stderr, "OPEN CSR ERROR %d\n",sqlca.sqlcode);
exit(-1);
}
/* Fetch Cursor */
for(;;)
{
EXEC SQL FETCH cur_emp INTO :emp_name, :job_title, :salary;

switch(sqlca.sqlcode)
{
case 0:
printf("Fetch Result : emp_name[%s], job[%s], sal=[%d]\n",
emp_name, job_title, salary );
continue;
case 1403: /* Not Found Data */
break;
default :
fprintf(stderr, "FETCH CSR ERROR %d",sqlca.sqlcode);
exit(-1);
```

```
 }
 break;
 }

 /* Cursor Close */
 EXEC SQL CLOSE cur_emp;
 /* Dynamic SQL */
 strcpy(dynamic_stmt, "DELETE FROM EMP WHERE EMPNO = :v1");
 EXEC SQL PREPARE sql_stmt FROM :dynamic_stmt;
 dynamic_emp_number = 10;
 EXEC SQL EXECUTE sql_stmt USING :dynamic_emp_number;
 /* Disconnect */
 EXEC SQL COMMIT WORK RELEASE;

 exit(0);
}
```

Altibase

```
#include <stdio.h>
#include <stdlib.h>
EXEC SQL BEGIN DECLARE SECTION;
char emp_name[21];
char job_title[21];
int salary;
int emp_number;
char uid[10];
char pwd[10];
char dynamic_stmt[120];
int dynamic_emp_number;
EXEC SQL END DECLARE SECTION;
int main(void)
{
 strcpy(uid, "SYS" );
 strcpy(pwd, "MANAGER");
 EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
 if ( sqlca.sqlcode != SQL_SUCCESS ) {
 fprintf(stderr, "DataBase Connect Error : [%d]!!!", sqlca.sqlcode);
 exit(-1);
 }
 /* INSERT */
 /* value setting */
 emp_number = 10;
 strcpy(emp_name, "ALTIBASE1");
 strcpy(job_title, "dba1");
 salary = 10000;
 /* INSERT DML */
 EXEC SQL INSERT INTO emp (empno, ename, job, sal)
 VALUES (:emp_number, :emp_name, :job_title, :salary);
 if ( sqlca.sqlcode != SQL_SUCCESS ) {
 fprintf(stderr, "DataBase Connect Error : [%d]!!!", sqlca.sqlcode);
 exit(-1);
 }
 emp_number = 20;
 strcpy(emp_name, "ALTIBASE2");
 strcpy(job_title, "dba2");
 salary = 20000;
 EXEC SQL INSERT INTO emp (empno, ename, job, sal)
 VALUES (:emp_number, :emp_name, :job_title, :salary);
 /* SELECT DML */
 emp_number = 10;
 EXEC SQL SELECT ename, job, sal INTO :emp_name, :job_title, :salary
 FROM emp
 WHERE empno = :emp_number;
 if ( sqlca.sqlcode != SQL_SUCCESS ) {
 fprintf(stderr, "Select Error : [%d]!!!", sqlca.sqlcode);
 exit(-1);
 }
 printf(" SELECT result : ename=[%s], job=[%s], sal=[%d]\n",
 emp_name, job_title, salary);
 /* UPDATE DML */
 emp_number = 10;
 salary = 2000;
 EXEC SQL UPDATE emp SET sal = :salary WHERE empno = :emp_number;
 if ( sqlca.sqlcode != SQL_SUCCESS ) {
 fprintf(stderr, "Update Error : [%d]!!!", sqlca.sqlcode);
 exit(-1);
 }
 /* Cursor Create */
 EXEC SQL DECLARE cur_emp CURSOR FOR
 SELECT ename, job, sal FROM emp;

 /* Cursor Open */
 EXEC SQL OPEN cur_emp;
 if(sqlca.sqlcode != SQL_SUCCESS ) {
 fprintf(stderr, "OPEN CSR ERROR %d\n",sqlca.sqlcode);
 exit(-1);
 }
 /* Fetch Cursor */
 for(;;)
 {
 EXEC SQL FETCH cur_emp INTO :emp_name, :job_title, :salary;

 switch(sqlca.sqlcode)
 {
 case SQL_SUCCESS:
 printf("Fetch Result : emp_name[%s], job[%s], sal=[%d]\n",
 emp_name, job_title, salary );
 continue;
 case SQL_NO_DATA: /* Not Found Data */
 break;
 default :
 fprintf(stderr, "FETCH CSR ERROR %d %s\n",
 sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
 exit(-1);
 }
 break;
 }

 /* Cursor Close */
 EXEC SQL CLOSE cur_emp;

 /* Dynamic SQL */
 strcpy(dynamic_stmt, "DELETE FROM EMP WHERE EMPNO = ?");
 EXEC SQL PREPARE sql_stmt FROM :dynamic_stmt;
 dynamic_emp_number = 10;
 EXEC SQL EXECUTE sql_stmt USING :dynamic_emp_number;

 /* Disconnect */
 EXEC SQL DISCONNECT;

exit(0);
}
```

Appendix C. Sample Programs

This Appendix includes the example programs used in the manual, schema, and basic data.

Execution of the Example Programs

Elements

$ALTIBASE_HOME/sample/APRE directory includes sample files, header files, schema creation files, and Makefiles.

The list of included files is as follows:

```
argument.sc
arrays1.sc
arrays2.sc
binary.sc
connect1.sc
connect2.sc
cursor1.sc
cursor2.sc
date.sc
delete.sc
dynamic1.sc
dynamic2.sc
dynamic3.sc
free.sc
indicator.sc
insert.sc
mc1.sc
mc2.sc
mc3.sc
mt1.sc
mt2.sc
pointer.sc
psm1.sc
psm2.sc
runtime_error_check.sc
select.sc
update.sc
varchar.sc
whenever1.sc
whenever2.sc
include/hostvar.h
schema/schema.sql
Makefile
```

Installation

When Altibase is installed, $ALTIBASE_HOME/sample/APRE directory will be automatically installed.

For more information about installation of ALTIBASE, see the *Installation User's Manual*.

Execution

The user can create an execution file using makefile saved under $ALTIBASE_HOME/sample/APRE directory.

Compile

```
make file_name
```

[Example1] An example of compiling of delete.sc example program.

```
make delete
apre -t cpp delete.sc
-----------------------------------------------
 APRE C/C++ Precompiler Ver 5.3.3.3
 Copyright 2009, ALTIBase Corporation or its subsidiaries.
 All rights reserved.
------------------------------------------------
g++ -D_GNU_SOURCE -W -Wall -pipe -D_POSIX_PTHREAD_SEMANTICS -D_POSIX_THREADS
-D_POSIX_THREAD_SAFE_FUNCTIONS -D_REENTRANT -DPDL_HAS_AIO_CALLS -g -DDEBUG -
fno-implicit-templates -fno-exceptions -fcheck-new -DPDL_NO_INLINE -
DPDL_LACKS_PDL_TOKEN -DPDL_LACKS_PDL_OTHER -c -I/home/trunk/work/altidev4/
altibase_home/include -I. -o delete.o delete.cpp
g++ -L. -g -L/home/trunk/work/altidev4/altibase_home/lib -o delete delete.o -
lapre -lodbccli -ldl -lpthread -lcrypt -lrt
```

Execution

./file_name

[Example2] An example of executing Delete, an execution file of delete.sc example program and checking the result.

```
./delete
<DELETE>
------------------------------------------------------------------
[Scalar Host Variables]
------------------------------------------------------------------
7 rows deleted
```

Table Information of the Example Programs

Purpose

Syntax and features of Altibase Embedded SQL Statement

Providing Script Files

Schema creation file

Provided as $ALTIBASE_HOME/sample/APRE/schema/schema.sql.

This file is to create the tables used in the manual and to insert the new example data. Therefore, to execute the examples in this manual, follow the provided files and practice exercises.

Schema

Function: Customer and Order control

Table: Employee, Department, Customer, Order, Product

Employee Table

Basic Key: Employee Number (ENO)

Record Size: 20

Column Name	Datatype	Description	Others
ENO	INTEGER	Employee Number	PRIMARY KEY
ENAME	CHAR(20)	Employee Name	NOT NULL
EMP_JOB	VARCHAR(15)	Title	
EMP_TEL	CHAR(15)	Tel. No.	
DNO	SMALLINT	Department No.	ASC
SALARY	NUMBER(10,2)	Monthly Salary	DEFAULT 0
SEX	CHAR(1)	Gender	DEFAULT ‘M’ NOT NULL
BIRTH	CHAR(4)	Birthday	NULL
JOIN_DATE	DATE	Employment Date	NULL
STATUS	CHAR(1)	Position	DEFAULT ‘H’

Department Table

Basic Key: Department Name (DNO)

Record Size: 8

Column Name	Datatype	Description	Others
DNO	SMALLINT	Department No.	PRIMARY KEY
DNAME	CHAR(30)	Department Name	NOT NULL
DEP_LOCATION	CHAR(9)	Department Location	
MGR_NO	INTEGER	Administrator No.	TTREE,ASC

Customer Table

Basic Key: Resident Registration No. (CNO)

Record Size: 20

Column Name	Datatype	Description	Others
CNO	BIGINT	Resident Registration No.	PRIMARY KEY
CNAME	CHAR(20)	Customer Name	NOT NULL
CUS_JOB	VARCHAR(20)	Occupation	
CUS_TEL	CHAR(15)	Tel. No.	NOT NULL
SEX	CHAR(1)	Gender	DEFAULT 'M' NOT NULL
BIRTH	CHAR(4)	Birthday	
POST	INTEGER	Postal Code	
ADDRESS	VARCHAR(60)	Address	

Orders Table

Basic Key: Ordering Date + Order No. (ONO , ORDER_DATE)

Record Size: 30

Column Name	Datatype	Description	Others
ONO	BIGINT	Order No.	PRIMARY KEY
ORDER_DATE	DATE	Order Date	PRIMARY KEY

Column Name	Datatype	Description	Others
ENO	INTEGER	Sales Clerk	NOT NULL, TTREE, ASC
CNO	BIGINT	Customer's Resident Registration No.	NOT NULL, TTREE, DESC
GNO	CHAR(10)	Product No.	NOT NULL, ASC
QTY	INTEGER	Order Quantity	DEFAULT 1
ARRIVAL_DATE	DATE	Expected Arrival Date	
PROCESSING	CHAR(1)	Order Status	DEFALT ‘O’

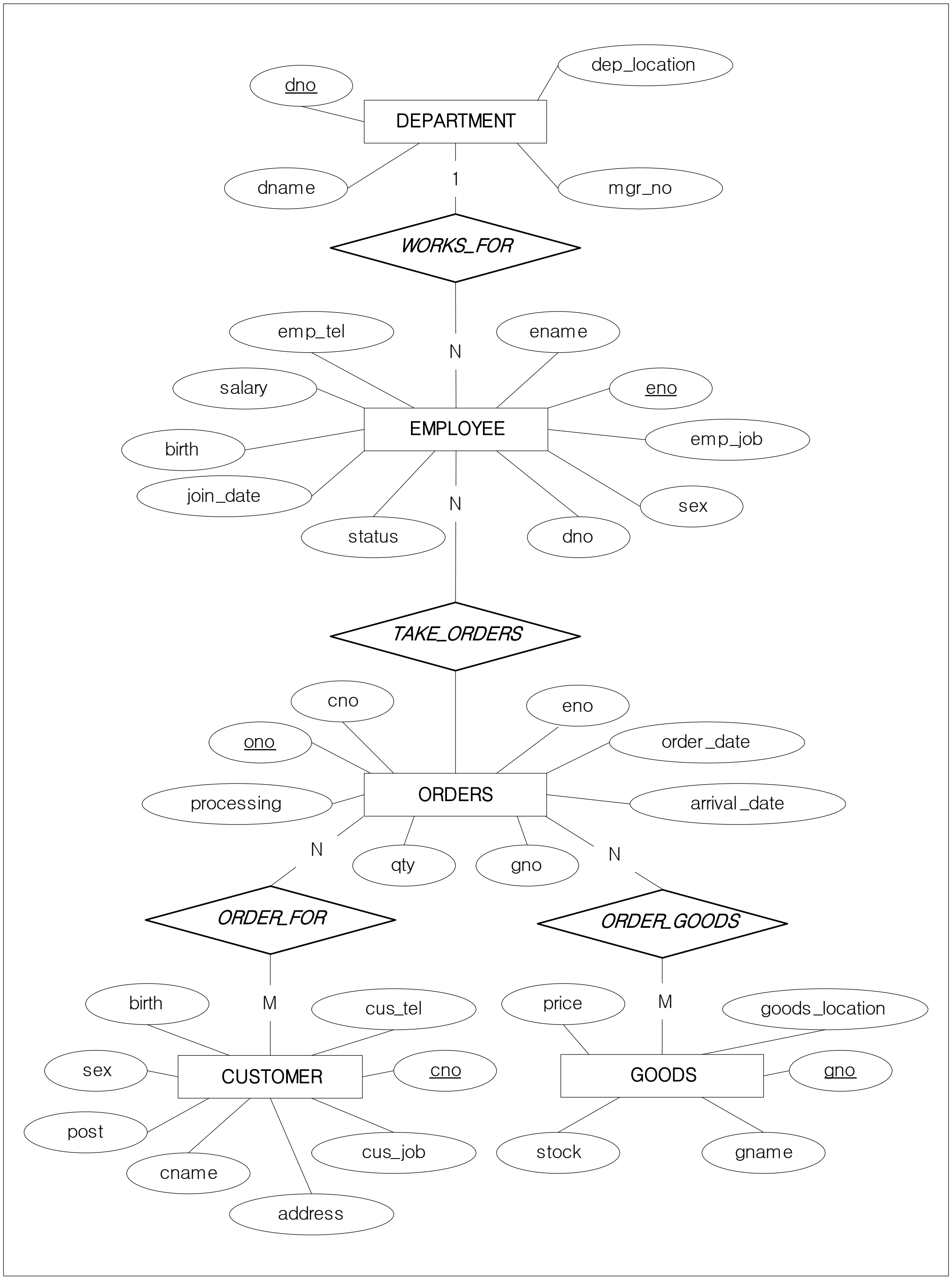
Goods Table

Basic Key: Product No. (GNO)

Record Size: 30

Column Name	Datatype	Description	Others
GNO	CHAR(10)	Product No.	PRIMARY KEY
GNAME	CHAR(20)	Product Name	UNIQUE NOT NULL
GOODS_LOCATION	CHAR(9)	Storage	
STOCK	INTEGER	Stored Quantity	DEFAULT 0
PRICE	NUMERIC(10,2)	Cost	

E-R Diagram


dep_location
dno
DEPARTMENT
dname
1
mgr_no
WORKS_FOR
emp_tel
ename
N
salary
eno
EMPLOYEE
birth
emp_job
join_date
N
sex
status
dno
TAKE_ORDERS
cno
eno
ono
order_date
ORDERS
processing
arrival_date
N
qty
gno
N
ORDER_FOR
ORDER_GOODS
birth
M
cus_tel
price
M
goods_location
sex
CUSTOMER
cno
GOODS
gno
post
cus_job
stock
gname
cname
address

Sample Data

Employment table

```
iSQL> SELECT * FROM EMPLOYEE;
EMPLOYEE.ENO EMPLOYEE.ENAME EMPLOYEE.EMP_JOB EMPLOYEE.EMP_TEL
----------------------------------------------------------------
EMPLOYEE.DNO EMPLOYEE.SALARY EMPLOYEE.SEX EMPLOYEE.BIRTH
----------------------------------------------------------------
EMPLOYEE.JOIN_DATE EMPLOYEE.STATUS
-----------------------------------------
1 EJJUNG CEO 1195662365
3002 M
 R
2 HJNO DESIGNER 113654540
 1500000 F 1219
1999/11/18 00:00:00 H
3 HSCHOI ENGINEER 162581369
1001 2000000 M 0226
2000/01/11 00:00:00 H
4 KSKIM PL 182563984
3001 1800000 M 0730
 H
5 SJKIM PL 1145582310
3002 2500000 M
1999/12/20 00:00:00 H
6 HYCHOI PROGRAMMER 197853222
1002 1700000 M 0822
2000/09/09 00:00:00 H
7 HJMIN MANAGER 175221002
4002 500000 M 0417
2000/01/24 00:00:00 H
8 JDLEE MANAGER 178829663
4001 M 0726
1999/11/29 00:00:00 H
9 KMLEE PLANER 165293668
4001 1200000 M 0102
2000/06/14 00:00:00 H
10 YHBAE PROGRAMMER 167452000
1003 4000000 F 0213
2000/01/05 00:00:00 H
11 MSKIM WEBMASTER 114553206
1003 2750000 M
2000/04/28 00:00:00 H
12 MYLEE SALESMAN 174562330
4002 1890000 F 0211
1999/12/14 00:00:00 H
13 KWKIM PM 187636550
1002 980000 M 1102
 H
14 KCJUNG PM 197664120
1003 2003000 M
 H
15 JHSEOUNG WEBMASTER 119556884
1003 1000000 M 1212
 H
16 JHCHOI MANAGER 195562100
1001 2300000 F 0509
 H
17 DIKIM PM 165293886
2001 1400000 M 1026
```

```
2000/05/07 00:00:00 H
18 CHLEE PLANER 1755231044
4001 1900000 M
2000/10/30 00:00:00 H
19 KMKIM SALESMAN 185698550
4002 1800000 M
2000/11/18 00:00:00 H
20 DIKIM SALESMAN 1154112366
4002 M
2000/11/18 00:00:00 H
20 rows selected.
```

Department Table

```
iSQL> SELECT * FROM DEPARTMENT;
DEPARTMENT.DNO DEPARTMENT.DNAME DEPARTMENT.DEP_LOCATION
------------------------------------------------------------------------
DEPARTMENT.MGR_NO
--------------------
1001 RESEARCH DEVELOPMENT DEPT 1 New York
16
1002 RESEARCH DEVELOPMENT DEPT 2 Sydney
13
1003 SOLUTION DEVELOPMENT DEPT Japan
14
2001 QUALITY ASSURANCE DEPT Seoul
17
3001 CUSTOMER SUPPORT DEPT London
4
3002 PRESALES DEPT Peking
5
4001 MARKETING DEPT Seoul
8
4002 BUSINESS DEPT LA
7
8 rows selected.
```

Customer Table

```
iSQL> SELECT * FROM CUSTOMER;
CUSTOMER.CNO CUSTOMER.CNAME CUSTOMER.CUS_JOB
---------------------------------------------------------------------
CUSTOMER.CUS_TEL CUSTOMER.SEX CUSTOMER.BIRTH CUSTOMER.POST
-------------------------------------------------------------------
CUSTOMER.ADDRESS
----------------------------------------------------------------
7308281201145 CHLEE ENGINEER
0514685282 M 0828 601033
DongGu Pusan
7712151345471 YSKIM DOCTOR
023242121 M 1215 121011
NamGu Taegu
7111111431202 DJKIM DESIGNER
023442542 M 1111 135010
Jigu Bank
7203052101114 JHPARK ENGINEER
022326393 F 0305 121758
Pusan University
7610121220475 BSYOUN WEBMASTER
0233452141 M 1012 121021
LG
```

```
6902091234567 IJLEE WEBPD
025743215 M 0209 136751
Samsung
7312252402221 JHCHOI PLANER
023143366 F 1225 156772
KT
7308011101115 HYCHOI PD
024721114 M 0801 135747
MapoGu Seoul
6002112417214 MYLEE DESIGNER
0512543734 F 0211 600033
Namsan Seoul
6208151724174 KSKIM
0516232256 M 0815 608703
SeoGu Taegu
7001011001001 LSPARK MANAGER
027664545 M 0101 142704
Sinchon Seoul
6709052101013 DHCHO BANKER
023343214 F 0905 152761
BangbaeDong Seoul
7912302114547 YDPARK ENGINEER
022320119 F 1230 153600
SinsaDong Seoul
7405081332014 DHKIM BANKER
024720112 M 0508 135740
KangnamGu Seoul
7506251122143 DKKIM MANAGER
0518064398 M 0625 606796
KangnamGu Seoul
7812251333044 SMCHO PLANER
027544147 M 1225 157703
SamsungDong Seoul
7610011000001 JHKIM
023543541 M 1001 157717
HabjungDong Seoul
7404192146506 JHKIM ENGINEER
024560207 F 0419 138701
YouidoDong SEoul
7312311515123 DJKIM
022371234 M 1231 138742
YouidoDong SEoul
7004052321123 DKHAN WEBMASTER
024560002 F 0405 135757
YeongdeungpoGu Seoul
20 rows selected.
```

Order Table

```
iSQL> SELECT * FROM ORDERS;
ORDERS.ONO ORDERS.ORDER_DATE ORDERS.ENO ORDERS.CNO
--------------------------------------------------------------------------
ORDERS.GNO ORDERS.QTY ORDERS.ARRIVAL_DATE ORDERS.PROCESSING
---------------------------------------------------------------------
11290007 2000/11/29 00:00:00 12 7111111431202
A111100002 70 2000/12/02 00:00:00 C
11290011 2000/11/29 00:00:00 12 7610011000001
E111100001 1000 2000/12/05 00:00:00 D
11290100 2000/11/29 00:00:00 19 7001011001001
E111100001 500 2000/12/07 00:00:00 D
12100277 2000/12/10 00:00:00 19 7610121220475
D111100008 2500 2000/12/12 00:00:00 C
12300001 2000/12/01 00:00:00 19 7308281201145
```

```
D111100004 1000 2001/01/02 00:00:00 P
12300002 2000/12/29 00:00:00 12 7712151345471
C111100001 300 2001/01/02 00:00:00 P
12300003 2000/12/29 00:00:00 20 7405081332014
E111100002 900 2001/01/02 00:00:00 P
12300004 2000/12/30 00:00:00 20 7506251122143
D111100002 1000 2001/01/02 00:00:00 P
12300005 2000/12/30 00:00:00 19 7203052101114
D111100008 4000 2001/01/02 00:00:00 P
12300006 2000/12/30 00:00:00 20 7912302114547
A111100002 20 2001/01/02 00:00:00 P
12300007 2000/12/30 00:00:00 12 7312252402221
D111100002 2500 2001/01/02 00:00:00 P
12300008 2000/12/30 00:00:00 20 7001011001001
D111100011 300 2001/01/02 00:00:00 P
12300009 2000/12/30 00:00:00 20 7312311515123
D111100003 500 2001/01/02 00:00:00 P
12300010 2000/12/30 00:00:00 19 7812251333044
D111100010 2000 2001/01/02 00:00:00 P
12300011 2000/12/30 00:00:00 20 7506251122143
C111100001 1000 2001/01/02 00:00:00 P
12300012 2000/12/30 00:00:00 12 7111111431202
E111100012 1300 2001/01/02 00:00:00 P
12300013 2000/12/30 00:00:00 20 6902091234567
C111100001 5000 2001/01/02 00:00:00 P
12300014 2000/12/30 00:00:00 12 6709052101013
F111100001 800 2001/01/02 00:00:00 P
12310001 2000/12/31 00:00:00 20 7506251122143
A111100002 50 2000/12/09 00:00:00 O
12310002 2000/12/31 00:00:00 12 6208151724174
D111100008 10000 2001/01/03 00:00:00 O
12310003 2000/12/31 00:00:00 20 7404192146506
E111100009 1500 2001/01/03 00:00:00 O
12310004 2000/12/31 00:00:00 19 7610121220475
E111100010 5000 2001/12/08 00:00:00 O
12310005 2000/12/31 00:00:00 20 7405081332014
E111100007 940 2001/01/03 00:00:00 O
12310006 2000/12/31 00:00:00 20 7712151345471
D111100004 500 2001/01/03 00:00:00 O
12310007 2000/12/31 00:00:00 12 7312311515123
E111100012 1400 2001/01/03 00:00:00 O
12310008 2000/12/31 00:00:00 19 7308281201145
D111100003 100 2001/01/03 00:00:00 O
12310009 2000/12/31 00:00:00 12 7610121220475
E111100013 500 2001/01/03 00:00:00 O
12310010 2000/12/31 00:00:00 20 6902091234567
D111100010 1500 2001/01/03 00:00:00 O
12310011 2000/12/31 00:00:00 19 7506251122143
E111100012 10000 2001/01/03 00:00:00 O
12310012 2000/12/31 00:00:00 19 7308281201145
C111100001 250 2001/01/03 00:00:00 O
30 rows selected.
```

Products Table

```
iSQL> SELECT * FROM GOODS;
GOODS.GNO GOODS.GNAME GOODS.GOODS_LOCATION GOODS.STOCK
-------------------------------------------------------------------------
GOODS.PRICE
--------------
A111100001 IM-300 AC0001 1000
78000
A111100002 IM-310 DD0001 100
```

```
98000
B111100001 NT-H5000 AC0002 780
35800
C111100001 IT-U950 FA0001 35000
7820.55
C111100002 IT-U200 AC0003 1000
9455.21
D111100001 TM-H5000 AC0004 7800
12000
D111100002 TM-T88 BF0001 10000
72000
D111100003 TM-L60 BF0002 650
45100
D111100004 TM-U950 DD0002 8000
96200
D111100005 TM-U925 AC0005 9800
23000
D111100006 TM-U375 EB0001 1200
57400
D111100007 TM-U325 EB0002 20000
84500
D111100008 TM-U200 AC0006 61000
10000
D111100009 TM-U300 DD0003 9000
50000
D111100010 TM-U590 DD0004 7900
36800
D111100011 TM-U295 FA0002 1000
45600
E111100001 M-T245 AC0007 900
2290.54
E111100002 M-150 FD0001 4300
7527.35
E111100003 M-180 BF0003 1000
2300.55
E111100004 M-190G CE0001 88000
5638.76
E111100005 M-U310 CE0002 11200
1450.5
E111100006 M-T153 FD0002 900
2338.62
E111100007 M-T102 BF0004 7890
966.99
E111100008 M-T500 EB0003 5000
1000.54
E111100009 M-T300 FA0003 7000
3099.88
E111100010 M-T260 AC0008 4000
9200.5
E111100011 M-780 AC0009 9800
9832.98
E111100012 M-U420 CE0003 43200
3566.78
E111100013 M-U290 FD0003 12000
1295.44
F111100001 AU-100 AC0010 10000
100000
30 rows selected.
```

Appendix D. FAQ

Frequently Asked Questions

About C/C++ Precompiler

Question 1

After executing my program, an incorrect data has been inserted into the table after the data was generated by the application created by the C/C++ precompiler.

Answer

When programming with C/C++ precompiler, the host variable must be a global variable. Please verify all your host variables is defined properly.

About Compiler Errors

Question 1

I can compile in Linux but I have a problem with compiling in Solaris Altibase. Execute "make clean."

make: Fatal error in

```
reader: /user/rttech/sjyu/altibase_home/install/src/makefiles/wrapper_m
acros.GNU, line 113: Unexpected end of line seen
 ---> ifeq ($(exceptions),0)
 override exceptions =
 endif # exceptions
```

When the following message appears, compiling will not be made.

OS: Solaris 2.7

Interface C/C++ Pre-compiler

Product: ALTIBASE-SPARC_SOLARIS_2.7-32bit-compat5-2.6.3-release.tar.gz

Answer 1

This error occurs because you did not use GNU MAKE. In most cases, GNU MAKE is installed in Linux. But in case of the SUN, the user needs to install this. After installing GNU MAKE, you should be able to compile the program.

GNU MAKE for SUN can be downloaded from www.sunfreeware.com.

Question 2

The Makefile in the sample directory that is created after installation of the program must have GNU Makefile. How can I use a general Makefile? Can I get a sample of a general Makefile?

Answer 2

Next environment : Following is an example of makefile compiling sun 2.8 64bit, $ALTIBASE_HOME/sample/sample1.sc.

```
COMPILE.c = /opt/SUNWspro/bin/cc -mt -fast -xarch=v9 -xprefetch=yes -
DPDL_NDEBUG -I/home2/hychoi/work/altibase_home/install/src/include -c
CC_OUTPUT_FLAG = -o LD = /opt/SUNWspro/bin/CC LFLAGS = -mt -xarch=v9 -
library=iostream,no%Cstd -L/opt/SUNWspro/SC5.0/lib/v9 -L/usr/lib/sparcv9 -
xprefetch=yes -fast -L/home2/hychoi/work/altibase_home/lib GOPT = INCLUDES =
-I$(ALTIBASE_HOME)/include -I. LIBDIRS = -L$(ALTIBASE_HOME)/lib LIBS=-xnolib
-Bdynamic -lthread -lposix4 -ldl -lkvm -lkstat -lsocket -lnsl -lgen -lm -lw -
lc -Bstatic -liostream -lCrun SRCS= OBJS=$(SRCS:.cpp=.o) BINS=sample1
apre=sample1.c SOBJS=$(SESS:.cpp=.o) %.c : %.sc apre $< all: $(BINS) sample1:
sample1.o sample1.c $(LD) $(LFLAGS) $(GOPT) $(INCLUDES) -o $@ sample1.o -
lapre -lodbccli $(LIBS) clean: -rm $(BINS) $(apre) *.o core *.class %.o: %.c
$(COMPILE.c) $(INCLUDES) $(CC_OUTPUT_FLAG) $@ $<
```

Question 3

A link error occurred in HP-UX upon library linking.

C compiler option is as follows:

```
cc +DA2.0W -I../include -I/user1/altibase/altibase_home/include/ -
I/user1/asn1/include -Ae -D_REENTRANT -DOAM -DSRC_LINE -DDEBUG -DDETAIL_DEBUG
-g -
c dbinit.c
```

Link the object files created later as follows:

```
cc +DA2.0W -o MKTDBD dbfunc.o main.o util.o dbif.o shm_msg.o file.o dbinit.o
dbresult.o -L/user1/altibase/altibase_home/lib -L/user1/main/KTSLEE/lib -
lcom -
lprice.1.1.0 -lodbccli -lapre -lxti -lpthread -lrt -ldld
```

Answer 3

From the execution result, it looks like you have used C compiler instead of C++ compiler for linking.

If you use C++ compiler, the system library is automatically added for compiling and linking. However, in case of using C compiler, you have to add a system library as follows:

```
LIBS += -ldl -lstd -lstream -lCsup -lm -lcl -lc
```

Change the link part as follows:

=====>

```
cc +DA2.0W -o MKTDBD dbfunc.o main.o util.o dbif.o shm_msg.o file.o dbinit.o
dbresult.o -L/user1/altibase/altibase_home/lib -L/user1/main/KTSLEE/lib -
lcom -
lprice.1.1.0 -lodbccli -lapre -lxti -lpthread -lrt -ldld -ldl -lstd -lstream
```

```
-lCsup -
lm -lcl –lc
cc +DA2.0W -o MKTDBD dbfunc.o main.o util.o dbif.o shm_msg.o file.o dbinit.o
dbresult.o -L/user1/altibase/altibase_home/lib -L/user1/main/KTSLEE/lib -
lcom -
lprice.1.1.0 -lodbccli -lapre -lxti -lpthread -lrt -ldld
```

Question 4

Errored for linking to gcc.

```
gcc -o OBJ/checkrep OBJ/checkrep.o -
L/home1/shkim/src/SK_DLR_v1.0.0/altibase/lib -
L/home1/shkim/src/SK_DLR_v1.0.0/ap/lib -ldlr -
lpara -lapre -lodbccli -lelf -lposix4 -lc -lxnet
Undefined first referenced
 symbol in file
kstat_close /home1/shkim/src/SK_DLR_v1.0.0/altibase/lib/li
bodbccli.a(idl.o)
gethostbyname_r /home1/shkim/src/SK_DLR_v1.0.0/altibase/lib/li
bodbccli.a(connection.o) (Symbol included in /usr/lib/libnsl.so.1 of hint
depending)
kstat_lookup /home1/shkim/src/SK_DLR_v1.0.0/altibase/lib/li
bodbccli.a(idl.o)
kstat_read /home1/shkim/src/SK_DLR_v1.0.0/altibase/lib/li
bodbccli.a(idl.o)
kstat_open /home1/shkim/src/SK_DLR_v1.0.0/altibase/lib/li
bodbccli.a(idl.o)
kstat_data_lookup /home1/shkim/src/SK_DLR_v1.0.0/altibase/lib/li
bodbccli.a(idl.o)
```

Answer 4

When using the Sun Compiler, SUN OS 2.7 and 2.6 refer to different system libraries for linking. It may depend on what you use for linking - CC, cc, or gcc.

1. 100% guarantee is not provided. But we recommend you to link using cc or CC. If the trouble consists, contact us.

2. We provide only -lodbccli and -lapre libraries. Others are all system libraries, and are used to link the archive saved in /user/lib (or a different directory.) Therefore, collision does not occur. Symbols that -lodbccli and -lapre refer must be stored in the system library to be linked.

When linking with gcc, we will compile libodbccli.a and libapre.a with gcc. For linking, it should link by creating system library to -lthread -lposix4 -ldl -lkvm -lkstat -lsocket -lnsl -lgen –l iostream -lCrun -lm -lw -lcx -lc.

Question 5

I understand that a server need to access the authentication server with an application written in C/CC++ precompiler to query the data of the MMDB in the authentication server. In this case, does the server must have C/C++ precompiler and have an independent execution module? Or does the library need to be in the authentication server? Distribute to each server, and link them to create an execution file. In the first case, every server that accesses the authentication server must have C/C++ precompiler, which is inconvenient. It is also inconvenient in terms of management. When creating a library and distributing it, can I link it and compile it in the server end (equipment, no OS)? Please provide a suggestion. I wonder what other methods are.

Please provide a makefile example.

Answer 5

If the OS is not correct, Altibase Client module (for compiling and linking) must be installed. In general, the same OS is used so that the program is compiled in the compiler machine or standard machine and only the execution models are copied to other servers. If the different OS are used, linker must be made in the corresponding machine. (See the system library.)

The makefile is as shown below. For linking, we provide the makefile with libapre.a and libodbccli. All basic information is included in altibase_env.mk.

Question 6

An error occurred while I was testing a simple source.

It looks like Altibase is conflicting with g++ in C++ standard IO part.

Answer 6

The following is an example of compiling with gcc:

```
apre -t cpp conn.sc
g++ -W -Wall -Wpointer-arith -pipe -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT -
fno-
implicit-templates -fno-exceptions -fcheck-new -I${ALTIBASE_HOME}/include -I.
-c -o conn.o conn.cpp
g++ -W -Wall -Wpointer-arith -pipe -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT -
fno-
implicit-templates -fno-exceptions -fcheck-new -I${ALTIBASE_HOME}/include -I.
-c -o main.o main.cpp
g++ -L. -L${ALTIBASE_HOME}/lib -o conn main.o conn.o -lapre -lodbccli -
lstdc++ -
lsocket -ldl -lnsl -lgen -lposix4 -lkvm -lkstat -lthread -lpthread
```

Question 7

Compile with CC compiler in the SUN OS.

```
> cc -DDEBUG source.c
```

After I updated Altibase to 2.6.4 and edited the source two days ago,

CFLAGS == -DDEBUG -O

I found that this option is not recognized while compiling the existing makefile.

The following defined variable is not defined on the source.

```
#if defined(DEBUG)
Variable
Variable..
#endif
```

Answer 7

Edit makefile as follows: (This is because some changes in the provided makefile.)

```
Change "CFLAGS == -DDEBUG -O" into "CC_OUTPUT_FLAG = -DDEBUG -O -o".
```

Question 8

I edited makefile, but it is not still working.

```
#include $(ALTIBASE_HOME)/install/altibase_env.mk
COMPILE.c = /bin/cc +DA2.0W +DS2.0W -DPDL_NDEBUG
CC_OUTPUT_FLAG = -c
LD = /opt/aCC/bin/aCC
LFLAGS = -L. +DA2.0W +DS2.0W -Wl,+vnocompatwarnings -L$(ALTIBASE_HOME)/lib
GOPT =
INCLUDES = -I$(ALTIBASE_HOME)/include -I.
LIBDIRS = -L$(ALTIBASE_HOME)/lib
LIBS=-lxti -lpthread -lrt -ldld
SRCS=
OBJS=$(SRCS:.cpp=.o)
BINS=altitest
apre=altitest.c
SOBJS=$(SESS:.cpp=.o)
%.o: %.c
$(COMPILE.c) $(INCLUDES) $(CC_OUTPUT_FLAG) $@ $<
%.c : %.sc
apre $<
all: $(BINS)
altitest: altitest.o altitest.c
$(LD) $(LFLAGS) $(GOPT) $(INCLUDES) -o $@ altitest.o -lapre -lodbccli
$(LIBS)
clean:
-rm $(BINS) $(apre) *.o core *.class
```

Answer 8

```
%.c : %.sc
apre $<
from above, change apre to apre.
```

Question 9

Precompiling is made, but an error keeps occurring when I try to link objects.

```
cc +DA2.0W -o MKTDBD dbfunc.o main.o util.o dbif.o shm_msg.o file.o dbinit.o
dbresult.o -L/user1/altibase/altibase_home/lib -L/user1/main/KTSLEE/lib -
lcom
lprice.1.1.4 -lodbccli -lapre -lxti -lpthread -lrt -ldld -ldl -lstd -lstream
-
lCsup -lm -lcl -lc
ld: Unsatisfied symbol "SESStmtCount" in
file /user1/altibase/altibase_home/lib/libapre.a[sesSqlcli.o]
1 errors.
*** Error exit code 1
Stop.
```

The host is HP L class, and HP-UX 11.0 is operating.

Altibase version is 2.6.3, and C/C++ precompiler version is 2.

Answer 9

Change the link order of –lodbccli

Question 10

I installed 2.4.1p1, but it does not match with the library version. Relink shell is required. (DEC4)

Answer 10

```
include $(ALTIBASE_HOME)/install/altibase_env.mk INCLUDES = -
I$(ALTIBASE_HOME)/include -I. LIBDIRS = -L$(ALTIBASE_HOME)/lib
ALTIBASE_OBJS=$(.cpp=.o) BINS_SERVER=ALTIBASE dbadmin checkServer createdb
destroydb killCheckServer restoredb shmutil BINS_CLIENT=isql iloader audit
apre all: $(BINS_SERVER) $(BINS_CLIENT) ALTIBASE:$(ALTIBASE_OBJS) /usr/lib/
cmplrs/cc/ld -o ALTIBASE -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/cmplrs/cxx -
L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/crt0.o /usr/lib/
cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/ALTIBASE.o -lmm -lqp -lsm -lid -lpd -
ltli -lrt -lpthread -lm -lcxxstd -lcxx -lexc -lc dbadmin:$(ALTIBASE_OBJS) /
usr/lib/cmplrs/cc/ld -o dbadmin -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/
cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/
crt0.o /usr/lib/cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/dbadmin.o -lmm -lqp -
lsm -lid -lpd -ltli -lrt -lpthread -lm -lcxxstd -lcxx -lexc -lc shmu-
til:$(ALTIBASE_OBJS) /usr/lib/cmplrs/cc/ld -o shmutil -L$(ALTIBASE_HOME)/lib
-rpath /usr/lib/cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/
lib/cmplrs/cc/crt0.o /usr/lib/cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/shmu-
til.o -lmm -lqp -lsm -lid -lpd -ltli -lrt -lpthread -lm -lcxxstd -lcxx -lexc
-lc createdb:$(ALTIBASE_OBJS) /usr/lib/cmplrs/cc/ld -o createdb -
L$(ALTIBASE_HOME)/lib -rpath /usr/lib/cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -
O4 -call_shared /usr/lib/cmplrs/cc/crt0.o /usr/lib/cmplrs/cxx/_main.o
$(ALTIBASE_HOME)/lib/createdb.o -lmm -lqp -lsm -lid -lpd -ltli -lrt -lpthread
-lm -lcxxstd -lcxx -lexc -lc destroydb:$(ALTIBASE_OBJS) /usr/lib/cmplrs/cc/ld
-o destroydb -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/cmplrs/cxx -L/usr/lib/
cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/crt0.o /usr/lib/cmplrs/
cxx/_main.o $(ALTIBASE_HOME)/lib/destroydb.o -lmm -lqp -lsm -lid -lpd -ltli -
lrt -lpthread -lm -lcxxstd -lcxx -lexc -lc checkServer:$(ALTIBASE_OBJS) /usr/
lib/cmplrs/cc/ld -o checkServer -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/
cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/
crt0.o /usr/lib/cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/checkServer.o
$(ALTIBASE_HOME)/lib/checkServerPid.o -lmm -lqp -lsm -lid -lpd -ltli -lrt -
lpthread -lm -lcxxstd -lcxx -lexc -lc killCheckServer:$(ALTIBASE_OBJS) /usr/
lib/cmplrs/cc/ld -o killCheckServer -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/
cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/
crt0.o /usr/lib/cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/killCheckServer.o
$(ALTIBASE_HOME)/lib/checkServerPid.o -lmm -lqp -lsm -lid -lpd -ltli -lrt -
lpthread -lm -lcxxstd -lcxx -lexc -lc restoredb:$(ALTIBASE_OBJS) /usr/lib/
cmplrs/cc/ld -o restoredb -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/cmplrs/cxx -
L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/crt0.o /usr/lib/
cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/restoredb.o -lmm -lqp -lsm -lid -lpd
-ltli -lrt -lpthread -lm -lcxxstd -lcxx -lexc -lc isql:$(ALTIBASE_OBJS) /usr/
lib/cmplrs/cc/ld -o isql -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/cmplrs/cxx -
L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/crt0.o /usr/lib/
cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/libisqlobj.a -lodbccli -lutil -lmm -
lqp -lsm -lid -lpd -ltli -lrt -lpthread -lm -lcxxstd -lcxx -lexc -lc
audit:$(ALTIBASE_OBJS) /usr/lib/cmplrs/cc/ld -o audit -L$(ALTIBASE_HOME)/lib
-rpath /usr/lib/cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/
lib/cmplrs/cc/crt0.o /usr/lib/cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/libau-
ditobj.a -lodbccli -lutil -lmm -lqp -lsm -lid -lpd -ltli -lrt -lpthread -lm -
```

```
lcxxstd -lcxx -lexc -lc iloader:$(ALTIBASE_OBJS) /usr/lib/cmplrs/cc/ld -o
iloader -L$(ALTIBASE_HOME)/lib -rpath /usr/lib/cmplrs/cxx -L/usr/lib/cmplrs/
cxx -g0 -O4 -call_shared /usr/lib/cmplrs/cc/crt0.o /usr/lib/cmplrs/cxx/
_main.o $(ALTIBASE_HOME)/lib/libiloaderobj.a -lodbccli -lutil -lmm -lqp -lsm
-lid -lpd -ltli -lrt -lpthread -lm -lcxxstd -lcxx -lexc -lc
apre:$(ALTIBASE_OBJS) /usr/lib/cmplrs/cc/ld -o apre -L$(ALTIBASE_HOME)/lib -
rpath /usr/lib/cmplrs/cxx -L/usr/lib/cmplrs/cxx -g0 -O4 -call_shared /usr/
lib/cmplrs/cc/crt0.o /usr/lib/cmplrs/cxx/_main.o $(ALTIBASE_HOME)/lib/liba-
preobj.a -lodbccli -lutil -lmm -lqp -lsm -lid -lpd -ltli -lrt -lpthread -lm -
lcxxstd -lcxx -lexc -lc clean: -rm $(BINS_SERVER) $(BINS_CLIENT) *.o core
*.class old: -mv ALTIBASE ALTIBASE.old -mv audit audit.old -mv checkServer
checkServer.old -mv checkipc checkipc.old -mv createdb createdb.old -mv dbad-
min dbadmin.old -mv destroydb destroydb.old -mv iloader iloader.old -mv isql
isql.old -mv killCheckServer killCheckServer.old -mv restoredb restoredb.old
-mv server server.old -mv apre apre.old -mv shmutil shmutil.old
```

Index